Spring 2020

I/O Master

Aaron Dubler
amd239@zips.uakron.edu

Corey Dye
cad140@zips.uakron.edu

Cameron Sinko
cjs196@zips.uakron.edu

Ian Glen
ijg6@zips.uakron.edu

Nikolaus Untch
nsu1@zips.uakron.edu

I/O Master

Senior Design Project Final Report

Aaron Dubler

Corey Dye

Ian Glen

Cameron Sinko

Nik Untch

Faculty Advisor:

Dr. French

24 April 2019

Table of Contents

List of Figures

List of Tables

Abstract

The I/O Master is an engineering tool designed to let users quickly and efficiently use their computer to interface with embedded devices and sensors. Using a general purpose design, the goal is to make it possible to implement any digital protocol on the I/O Master. With the use of onboard level shifting, differential communication components, peripheral DMA through GPIO and a USB 2.0 High Speed computer interface, the I/O Master is able to implement such a general purpose design with initial implementation of five protocols to verify the design's capabilities. The implementation of a computer-based GUI and underlying framework allows users to easily use the I/O Master with additional flexibility to write custom programs for it. While many characteristics of components were considered for the design, the design lacks quantified information relating to propagation delay of components and response time for some of the electrical safety components. Engineers can use the I/O Master to make the early phases of design faster when there is a need to interface with sensors for testing and analysis. The design of the I/O Master allows for additional protocols to be implemented in the future with only limited changes needed in the software.

Major features include:

- Implementations to read and write UART, SPI, I2C, SWD, and RS-485 protocols
- Designed to support any single-ended or differential protocol using four or less I/O lines from -15v to 15v and up to 10 MHz
- Computer based GUI that lets engineers use the I/O master without needing to read documentation on the tool

- Can operate off of USB power or external 12 V power supply
- Safety from user error with over-voltage and over-current protection

1. Problem Statement

1.1 Need

The need for a general purpose protocol interfacing device comes from the diverse number of products and protocols from companies across multiple industries, such as cars, microcontroller kits, hard drives, and LCD displays. Many products each have a unique protocol to interface with, and often require an expensive, specific purpose, proprietary tool that only works with a few products. A general purpose tool that is cost effective and has the ability to interface with many different products is needed. This eliminates the need for several expensive, specific devices and the learning curve that comes with using all of them. There are currently general purpose tools, such as the Bus Pirate, but they rely heavily on software solutions and designs that are over a decade old. (NU)

1.2 Objective

The objective is to create a general purpose tool that is capable of implementing a variety of protocols and switching between them using reconfigurable hardware. Rather than implementing this as a software focused solution, the I/O Master implements a more modern hardware platform that can support a wider range of voltages, frequencies, and signal types with great accuracy. The improved hardware platform is able to support more complex protocols, in addition to what current devices support. While the I/O Master is designed to be general purpose, during development focus will be on the implementation of a few key protocols that showcase the new hardware's more advanced capabilities. As with similar tools, the responsibility of the device is

to send and receive data frames on a low level, and not to identify, parse, or make sense of what the data means. (NU)

1.3 Background

There are a wide range of products that computer and electrical engineers develop, analyze, and interface with, from automobiles to microcontroller kits. As more products come to market utilizing digital communication, the number of different communication protocols increases. While each protocol is unique, most share many similarities basic interfacing protocols, such as I^2C and SPI. (CS)

I^2C and SPI are two examples of common communication protocols used in devices to talk to I/O peripherals. Serial Peripheral Interface (SPI) uses four wires: a clock signal (SCLK), a slave select signal (SS), a master to slave data signal (MOSI), and a slave to master data signal (MISO). In this protocol, there is one master and all the other connected devices are slaves. The master sends a signal via the slave select line to the slave it wishes to communicate with. It then establishes a clock frequency to use based on what the slave requires. There is no limit for the speed of the communication. There are also no measures in place to ensure that the message that was sent was ever received. Other outside implementations are needed to ensure accurate and complete data transfer. [1] (CD)

Inter-Integrated Circuit (I^2C) is slightly more complex but more reliable. This protocol only uses two lines for communication, a line for a clock and the other for serial communication. Unlike SPI, I^2C does not have a set master for the bus but whichever device initiates the data transfer is determined to be the master and all other devices become slaves. An advantage of I^2C is that it

has a built in method for data control. 7 bits are used for I^2C addressing. This means that the protocol is limited to as many devices as 7 bits can use. However, there are workarounds used to get more address bits, such as 10 bit addressing. [1] (CD)

The fundamental component of a protocol are the digital logic levels. Most digital devices are driven by analog devices. This allows systems to create two distinct voltage logic levels most commonly referred to as logic high and logic low. The typical system uses either +5V or +3.3V as a logic high and 0V (ground) as a logic low. In practical applications, logic high and logic low are usually defined as ranges. [2] (NU)

These types of systems can be wired as either active-high or active-low. In an active high application, the active-high pin is connected to the high voltage. Conversely, in an active-low application, the pin is connected to the ground. [2] (NU)

There are additional logic levels besides logic high and logic low. The Z state is a tristate logic state that is created using high impedance where the output is disconnected. This is commonly used in applications where multiple transmitters are used. The X state is the fourth state. This is referred to as the hold state, which is most commonly used for setting initial states or idle states. [2] (NU)

Logic families provide standardized voltage levels for a logic high or logic low. This allows compatibility between several different circuits as long as the circuits share the same standard logic family. Single-ended logic families use a known voltage in reference to ground. There are four levels in this logic family: output high level voltage, output low level voltage, input high level voltage, and input low level voltage. To account for error and noise, most systems have a

buffer between voltages to clearly separate the different states. Differential logic families do not use a ground as a reference. Instead, the signal is the difference between two known voltages. A logic low for differential logic families can be defined by the signal having a value less than the threshold value. [2] (NU)

Built upon these common protocols are more specialized protocols used for specific tasks, such as programming microcontrollers. The vast amount of specialized protocols that exist is the main motivation behind the development of the I/O Master. Often, manufacturers use a specific protocol only for a certain product line of devices, and as a result specific, single-purpose devices must be purchased and typically designed by the manufacturer in order to work with these protocols. (IG)

Microchip, the company behind PIC microcontrollers, implements a protocol called In-Circuit Serial Programming (ICSP) which is used for programming their microcontrollers. This protocol is very similar to SPI, as Microchip states, “a 2-wire SPI is used and data transmissions are half-duplex, a simple protocol is used to control the direction of PGDx” [3]. This means that while ICSP uses SPI hardware, it is slightly augmented. (CS)

Microchip offers a device known as the PICkit programmer: a proprietary, purpose-built device that is designed to interface between a computer and a PIC microcontroller over ICSP. Microchip creates a specific purpose device because it is cheaper to produce, leads to more sales, and restricts developers to owning tools that only work for their products. Microchip can sell these devices with high profit margins, and given that it can only perform one task, leaves the end user with an unfavorable device. While the PICkit is considered a proprietary device, Microchip fully

documents how the ICSP protocol works and how to implement it to program PIC microcontrollers. (CS)

Microchip's programming specification helps break down what goes into the protocol. The ICSP specification is broken down into various sections including signal frequency and voltage requirements, input and output data lines, length and layout of frames, a table of commands, and flowcharts for communication processes [3]. Some of these parts are better implemented in hardware, while others are better implemented in software. (CS)

This need to purchase many single-protocol devices in order to use a variety of protocols is the motivation behind designing a universal communication device. By designing more general purpose hardware and software that focuses on the basics of what makes up a protocol, the I/O Master could then be configured in a way that emulates the operation of a PICkit device, yet still be adaptable to other interfaces and protocols. (CS)

As a solution to the issues that arise from purpose-built devices, there are a few existing general purpose interface devices. Among these include the Bus Pirate and the GoodFET. These devices focus on being a cost effective way to interface with a variety of protocols. However, the functionality of these devices are limited. (CS)

The original design of the Bus Pirate dates back to 2004, and since then there have been four major hardware revisions [4]. The GoodFET device was developed initially around 2009 and a few variations have been produced in the following years. Both the Bus Pirate and GoodFET heavily rely on bit banging, a software technique used to emulate protocols without requiring specialized hardware [5]. While extremely flexible, bit banging depends heavily on the clock

signal of the microcontroller, and the results are not perfect. In addition, digital voltage levels are limited to what the microcontroller supports. Performance issues are major factors with these devices, “Another drawback of GoodFET is performance. Interfaces such as SPI are implemented by bit-banging and do not take advantage of higher speed peripherals available on the microcontroller” [5]. (CS)

Essentially, these devices utilize hardware that is too generalized to succeed in more complex applications. At the time, it was very expensive to add reconfigurable hardware. The addition of another hardware component is necessary to implement more complex protocols, such as differential signal protocols, USB, or higher frequency SPI. By adding an FPGA as an intermediary between the microcontroller and the device I/O, hardware can do the low level work of reading and writing signals into a fast buffer. This yields the ability to better adapt to different voltage levels and frequencies, but still offer the flexibility to parse signals through embedded software. When the Bus Pirate and GoodFET were designed, FPGAs were still very expensive, but the cost has gone down in recent years, making the inclusion of one a viable option for such a device. However, microcontrollers are significantly faster than they once were and the additional cost of a much faster microcontroller may be sufficient and more cost effective than a slow microcontroller and FPGA. (CS)

In designing the I/O Master, many existing technologies can be used to help solve various challenges in each component of the system. Hardware processing of protocol data must be reconfigurable during operation so protocols can be switched between seamlessly. The physical I/O interface of the device must be able to adapt to the voltage levels of the current protocol in

use. As a development tool, the I/O Master must be robust. Some protection should exist from user error or failure of an attached device. Computer control, and especially for wireless control, must be standardized and well supported so that the end user can easily use the device. (IG)

Many existing products have similar focus as the I/O Master. US Patent 7,099,438 describes a device able to simultaneously communicate via different interface protocols to analyze and record data for testing of telecommunications equipment. These protocols include time-division multiplexing (TDM), internet-protocol (IP), and various telephony application program interfaces [6]. (NU)

A system that must support a variety of protocols needs a solution on how to provide support through hardware. Many applications often include specialized circuits for each communications interface, which leads to increased cost, size, and development time as the number of interfaces increases. One example where this has become an issue is in the maintenance of flight computers for military aircraft. As the need for updating software has increased, as well as the number of specialized avionics computers has grown, “large boxes containing custom circuits were created ... this worked well as long as there were only a few computers to program” [7]. With the introduction of F-15 and F-16 aircraft and their many computers, the military turned to solutions that utilized programmable devices such as programmable logic devices (PLDs) and field programmable gate arrays (FPGAs) to replace much of the custom hardware. PLDs and FPGAs allow for reprogrammability by providing logic blocks that can be interconnected to perform the same operation as discrete logic circuits or application-specific integrated circuits (ASICs). PLDs have the added benefit of retaining their configuration even when power is not applied, while

FPGAs use random-access memory (RAM) to store configuration data and must be reprogrammed during power up. FPGAs tend to provide more logic blocks and functionality compared to PLDs and can be reconfigured without having to remove power. In the case of military avionics computers, an FPGA was utilized to develop the Universal Memory Loader/Verifier (UMLV) and later the Personal Computer Memory Loader/Verifier (PCMLV) devices, which allow communication with many different avionics computers over standard interfaces (RS-232, RS-422, and IEEE-488) [2]. Given the limitation that the FPGA I/O lines only supported 5V TTL level digital signals, additional voltage level shifting hardware was paired with the FPGA to support the +/-12V signals of RS-232. In essence, the use of programmable devices such as FPGAs allowed the defense industry to support a large number of specialized devices in a cost effective manner. (IG)

Unlike many electronics applications, a universal device with general purpose I/O lines has a greater possibility of being subject to harmful electrical signals, as the hardware is exposed to the end user. As such, it is important to include in the design a portion responsible for protecting the device in the event of failure or human error. US Patent 7,649,726 explores a method of circuit protection using a variety of techniques to protect general purpose I/O lines from “destructive electrical conditions, such as electrostatic discharge (ESD), over-voltage, and over-current” [8]. The circuit protection system consists of two protection circuits and a bidirectional I/O buffer. For over-voltage protection, zener diodes are used as a voltage clamping device and are paired with a current limited device (such as a positive-temperature coefficient, or PTC, resistor) to limit the power dissipation in the diodes. For overcurrent protection, a low value resistor is used to limit I/O current under normal operating conditions. When the operating conditions of the

resistor are exceeded, an active power dissipation circuit takes over in order to limit current. By combining basic protection circuits, such as clamping diodes, current limiting devices, and filters, an I/O interface of the I/O Master can be protected from a variety of electrical situations that would otherwise cause damage. (IG)

1.4 Marketing Requirements

1. The I/O Master will allow electrical and computer engineers with minimal experience to configure, read, and write data using an intuitive user interface.

2. The I/O Master will implement various common protocols used to communicate with a variety of microprocessors, memory ICs, and sensors by utilizing reconfigurable hardware.

3. The I/O Master will be reconfigurable on the fly without needing to be reprogrammed with different firmware.

4. The I/O Master will be robust and will not be damaged by user error.

5. The I/O Master will support fully configurable voltage/logic levels, not restricted to specific standards.

6. The I/O Master will have the ability to power the target device for the user's convenience of not having to use an additional power supply.

2. Engineering Analysis

2.1 Microcontroller vs. FPGA Cost-Benefit Analysis

It is known from background research that the inclusion of an FPGA in the design adds functionality to the device. However, it was not heavily considered that modern microcontrollers are offered with high speed operating frequencies, which complements the core benefit of the FPGA. Therefore, it is necessary to determine whether solely a high speed microcontroller or a low speed microcontroller paired with an FPGA is a better design choice.

Looking into high speed microcontrollers, research begins with examining the clock speeds of the highest-end microcontrollers from Microchip, Atmel, and STMicroelectronics and the resulting cost. Microchip sells the PIC32MZ series, which clocks at up to 250 MHz, and costs up to $14 to purchase a single unit. Atmel's product line has the SAMS70 series, which clocks up to 300 MHz and costs up to $14 to purchase a single unit. STMicroelectronics has the H7 series, which clocks up to 480 MHz and costs up to $15 to purchase a single unit.

All of these microcontrollers are able to operate at well over 200 MHz, which is significantly faster than the Bus Pirate's clock speed of 32 MHz, some over ten times the speed. The Bus pirate could interface protocols of up to 1 MHz, but I^2C, one of the target protocols for the I/O Master, has a high speed mode that operates at 3.4 MHz. In addition, RS-485 can operate at up to 10 MHz.

Based on the Nyquist-Shannon sampling theorem, in order to sample 10 MHz signals, it is necessary to have two times the signal frequency, so 20 MHz is the minimum to read the signal.

However, given that additional clock cycles are required to parse and route the data, a microcontroller frequency higher than 20 MHz is necessary. With a 200 MHz microprocessor, each bit of data would be allotted a maximum of ten clock cycles of processing. Microcontrollers with data bus sizes of 32 bits are being considered, so it would be most efficient to process data in chunks of 32 bits because this microcontroller can run an instruction on a 32 bit chunk of data in one cycle. If ten clock cycles of processing are allotted for each bit of data, Then for every 32 bit chunk that gets processed, there are $32 \cdot 10 = 320$ clock cycles available to process the data. This is more than enough to be able to read, process, and write data at 10 MHz without having the microprocessor be a bottleneck.

With a microcontroller and an FPGA, the FPGA can buffer data into blocks and send it out in parallel to the microcontroller. Suppose a microcontroller is used with approximately a 200 MHz clock and a parallel I/O bus that is 8 bits wide, and suppose the FPGA is significantly faster than the microcontroller. Now, data can be read at eight times the speed since each microcontroller clock can read in eight bits in parallel. As a result, data can be read at approximately 80 MHz. Since data is being streamed into the microcontroller at eight times the speed, the 320 clock cycles to process each 32 bit chunk of data would be diminished to $320 / 8 = 40$ clock cycles for each 32 bit chunk of data. The 40 clock cycles should be enough to move data to another interface, but is significantly less than 320. This means that the microcontroller would need to be better optimized in order to make routing data at 80 MHz achievable.

With this idea in mind, if a microcontroller is scaled down to 1/8th of the 200Mhz frequency to 25 MHz, then a microcontroller could be paired with an FPGA to achieve support for interfacing

1/8th of the 80 Mhz frequency, 10 MHz. STMicroelectronics has a STM32F1 series controller which can clock in at 24 MHz, which is close to 25 MHz. These microcontrollers sell for about 3 dollars each. To achieve a comparable total cost, a $12 dollar FPGA would need to be implemented. However, considering the need for additional parts to interface and power the FPGA, an FPGA under $10 in cost would be necessary. Lattice Semiconductor offers an FPGA for $9.50 which features enough I/O and a 400 MHz clock. This would be fast enough to interface 10 MHz signals for a similar price point of $15. However, there are downsides.

With an FPGA, moving data is more complex as it has more steps in the pipeline. Design of the FPGA logic blocks would also need to be done, which is a significant addition to the project. Also, the FPGA, small microcontroller combination allows for less clock cycles to process each piece of data (40 clock cycles is less than 320 clock cycles), which is less ideal. In all, this is a more complex design with less potential for the same cost.

At higher frequencies, such as beyond 50 or 100 MHz, the addition of an FPGA would be necessary. However, with today's high powered microcontrollers, achieving a consistent data throughput of 10 MHz is achievable without the use of an FPGA. (CS)

2.2 Microcontroller Selection

The most important design decision that influences the direction of the project and future design decisions is the selection of a microcontroller family. Since the implementation of the I/O Master utilizes a high-speed microcontroller to generate digital signals with a software-defined approach, certain microcontroller features are critical. From a signal speed prospective, in order to account for Nyquist frequency, memory transfer overhead, USB communication overhead, and additional bottlenecks, the CPU/peripheral clock frequencies must be significantly faster than the maximum speed of the signal (at least one order of magnitude, i.e. at least 100 MHz). From a data flow perspective, direct memory access (DMA) controller support and on-chip USB support are important features that make it easier to implement a high-speed system, Finally, pin count, chip packaging, and estimated cost should be considered to ensure a practical physical design in the context of a senior design project.

This section compares various high-end 32 bit microcontrollers, specifically the Microchip PIC32MZ, Atmel SAMS70, STMicro STM32F7, and STMicro STM32H7 families. The comparison is summarized in Table 1 found at the end of this section. There are likely lower-spec microcontroller families that would work for this project, however by choosing one that is beyond the requirements, the risk of having to switch microcontrollers midway through development is minimized. In the long term, it is certainly less involved to downsize to a smaller microcontroller for the final prototype than to a larger microcontroller -- and potentially one from a different family.

The Microchip PIC32MZ family is a line of high-end 32-bit microcontrollers with a MIPS32 core. Advanced features are available including a floating point unit (FPU), digital signal processor (DSP), graphics processor, and integrated DRAM. The CPU clock frequency can be up to 252 MHz with up to 2 MB of flash memory and up to 640 kB of SRAM. The DMA controller provides 26 channels with dedicated channels for certain peripherals, such as the integrated USB interface, which supports up to USB 2.0 speeds (480 Mbit/s theoretical). Various chip packages are available with up to 288 pins, more than enough for the I/O requirements of this project. Microchip products typically have a very good supply chain and are readily available. Based on normally stocked products at Digikey, looking at only hand-solderable chip packages, these microcontrollers are available for $7.90~$20.56 in single quantities and for $6.56~$17.07 in quantities of 100.

The Atmel SAMS70 family is a line of high-end 32-bit microcontrollers with an ARM Cortex-M7 core. These microcontrollers are available with a floating point unit (FPU), integrated USB 2.0 (480 Mbit/s theoretical), and integrated Ethernet. The CPU clock frequency can be up to 300 MHz with up to 2 MB of flash memory and up to 384 kB of SRAM. Since the CPU is an ARM core, it shares the same instruction set with all other ARM Cortex microcontrollers, allowing for greater portability if there is a need to switch microcontroller families. The integrated DMA controller has up to 24 DMA channels with a dual AHB master interface. Various packages are available with up to 144 pins. Based on normally stocked products at Digikey, looking at only hand-solderable chip packages, these microcontrollers are available for $7.71~$12.80 in single quantities and for $6.68~$10.54 in quantities of 100.

The STmicro STM32F7 family is a line of high-end 32-bit microcontrollers with an ARM Cortex-M7 core. These microcontrollers are available with advanced features including a floating point unit (FPU), graphics accelerator, and integrated USB 2.0 (480 Mbit/s theoretical). The CPU clock frequency can be up to 216 MHz with up to 2 MB of flash memory and 512 kB of SRAM. Similar to the Atmel SAMS70 family, the use of an ARM Cortex CPU allows for greater code portability. Two general purpose DMA controllers with eight channels are available. Based on normally stocked products at Digikey, looking at only hand-solderable chip packages, these microcontrollers are available for $3.97~$16.26 in single quantities and for $2.92~$12.62 in quantities of 100.

STmicro also offers a higher-end version of the STM32F7 family with the option of a dual-core CPU, known as the STM32H7. Dual-core versions include a Cortex-M4 core along side the primary Cortex-M7 core. CPU clock frequency is increased up to 480 MHz with up to 2 MB of flash memory and up to 1 MB of SRAM. Since the STM32H7 family shares many similarities with the STM32F7 family and both utilize an ARM Cortex-M7 core, it would be very easy to move between the two families if necessary. Some packages of the STM32H7 are pin compatible with certain STM32F7 and STM32F4 chips. A high-speed master direct memory access controller (MDMA) is provided along with two peripheral-connected DMA controllers and an additional general-purpose DMA controller. The high-speed MDMA controller has its own AHB master interface with 16 channels and is capable of any kind of transfer, including to/from memory, to/from peripherals, or a combination, a feature not normally found in standard DMA controllers. Based on normally stocked products at Digikey, looking at only hand-solderable chip

packages, these microcontrollers are available for $13.10~$14.89 in single quantities and for $10.16~$11.55 in quantities of 100.

Table 1: Comparison of Microcontroller Families

	PIC32MZ	**SAMS70**	**STM32F7**	**STM32H7**
CPU Frequency	252 MHz	300 MHz	216 MHz	480 MHz
Flash Memory	2 MB	2 MB	2 MB	2 MB
SRAM	640 kB	384 kB	512 kB	1 MB
DMA	- Single controller - 26 channels	- Single controller - 24 channels - Dual interface	- Two controllers - 8 channels each	- Four controllers - MDMA - 16 channels each
USB	USB 2.0	USB 2.0	USB 2.0	USB 2.0
Pin Count	288	144	208	208
Price for Qty. 1	$7.90 ~ $20.56	$7.71 ~ $12.80	$3.97 ~ $16.26	$13.10 ~ $14.89
Price for Qty. 100	$6.56 ~ $17.07	$6.68 ~ $10.54	$2.92 ~ $12.62	$10.16 ~ $11.55

Based upon the above analysis, the STM32H7 appears to be best suited for this project. The advanced DMA features available with this family could be useful in developing a high performance solution for generating software-defined digital signals. In addition, its similarity to the STM32F7 and the rest of STmicro's STM32 microcontrollers make it a flexible choice that allows the project to be scaled down to a smaller and more cost effective microcontroller in the future if needed. While the PIC32MZ or SAMS70 seems viable given its low cost, availability, and popularity, these benefits simply do not make up for the lack of advanced features that the STM32H7 family offers. From a prototyping perspective, the significantly faster CPU clock, larger amount of SRAM, and large amount of GPIO pins allow for fast development and minimizes the risk of having to switch families mid-project. To make development even faster, STmicro provides a low-cost development board with a built-in programmer for the STM32H7

chip so software development can progress independently of the hardware development of this project. (IG)

2.3 Hardware Design

2.3.1 Protocol Electrical Requirements

The engineering requirements for the hardware design of the I/O Master must be chosen such that the hardware is as general purpose as possible. In order to keep the scope of the project narrow, the I/O Master is specifically designed to support the following protocols: RS-232, inter-integrated circuit (I^2C), serial peripheral interface (SPI), serial wire debug (SWD), and RS-485. These protocols are selected because they cover a wide variety of electrical requirements, and a hardware design that supports these protocols should be general purpose enough that additional protocols can be supported with only some changes to the I/O Master's software. By examining these protocols, target values can be determined for the voltage range, maximum current, and maximum frequency of the I/O Master's digital interface. A summary of the electrical requirements of these protocols can be found in Table 2 at the end of this section.

RS-232 is a type of serial communication protocol known as a universal asynchronous receiver/transmitter (UART). It is an example of a very common protocol with large voltage swings and high current requirements. Two signals are needed for basic RS-232 communication: a transmit signal (TX) and receive signal (RX). Unlike the other protocols that the I/O Master will implement, with the exception of RS-485, RS-232 is a fully asynchronous communication protocol where either device can send data at any time and there is no separate clock signal for synchronization. It is also not a bus system like SPI or I^2C, as the pair of signal lines is intended

to only be between two devices. The EIA-232 standard defines the protocol, including the required signals, electrical requirements, and format for sending bytes. The transmitter voltage levels are between -5V and -15V for a LOW-state and +5V to +15V for a HIGH-state. The receiver voltage levels are between -3V and -15V for a LOW-state and +3V to +15V for a HIGH-state. [12] The required output current for the transmitter depends on the line load capacitance (which is proportional to the signal line length) and transmission speed. Figure 1 shows the relationship between output current and line load capacitance for various transmission speeds.

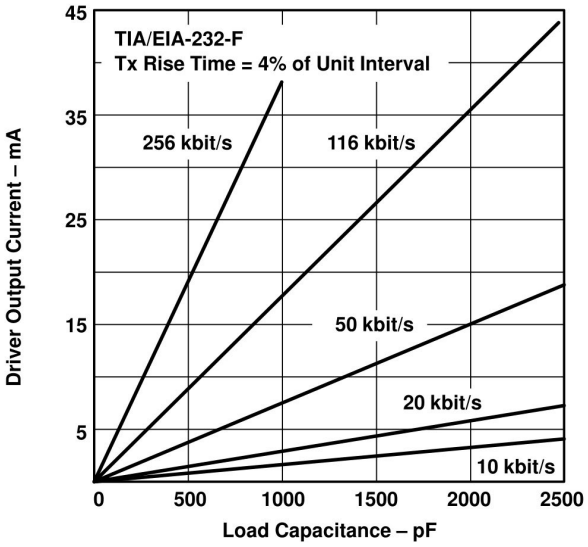


Figure 1: RS-232 Output Driver Current vs. Line Load Capacitance [12]

Based on the graph, drive currents up to 45 mA may be required when long lines are used. Common transmission speeds include 9600 bits/sec and 115200 bits/sec, with some systems transmitting up to 2 Mbits/sec.

I^2C is a two-wire protocol for communication between a master device and many slave devices connected to a single bus. The physical bus consists of two wires—a clock signal (SCL) and a bidirectional data signal (SDA). I^2C is a synchronous protocol where the master device controls the clock signal, which in turn defines when slave devices are able to transmit and receive data. The I^2C protocol is standardized by NXP Semiconductors and defines electrical specifications, transmission speeds, and data formats. Since communication is bidirectional, pull-up resistors are required on both SCL and SDA signal lines and each transmitter switches between a floating state and a logic-LOW state. The signal line returns to an idle logic-HIGH state when neither the master nor slave devices are driving the lines. The electrical requirements depend on the supply voltage of the master and slave devices. Most common devices use either a 3.3V or 5V supply voltage. Assuming a 5V supply voltage, the transmission voltage levels from the master to a slave device are 0V to 0.4V for the LOW-state and 3.5V to 5.5V for the HIGH-state. The receiver voltage levels from a slave device to the master are -0.5V to 1.5V for the LOW-state and 3.5V to 5.5V for the HIGH-state. Three transmission speeds are available: 100 kbit/s (standard mode), 400 kbit/s (fast mode), and 1 Mbit/s (fast mode plus). Additional modes are available for even higher speed communication up to 5 Mbit/s. For standard mode and fast mode, the transmitter must be able to drive up to 3mA, while for fast mode plus the transmitter must be able to drive up to 20mA. [10]

SPI is a four-wire protocol similar to I^2C in the sense that it allows many devices to communicate over a single bus. A minimum of four wires are required, one for the clock signal (SCK), one for data sent from master to slave (MOSI), one for data sent from slave to master (MISO), and a chip select line for each slave device (CS). Like I^2C, the master determines when communication

occurs. The chip select line determines which device on the bus is currently communicating. No additional resistors are required on MOSI, MISO, SCK, or CS signal lines since all outputs are push-pull type. It is important to note that SPI is a de facto standard: there isn't a single, defined standard, however SPI is widely used and many companies publish generally agreed upon electrical characteristics. Typical SPI devices operate with a supply voltage V_{DD} up to 5V. Assuming a 5V supply voltage, the transmission voltage levels from the master to a slave device are 0V to 0.5V for the LOW-state and 4.5V to 5V for the HIGH-state. The receiver voltage levels from a slave device to the master are -0.3V to 1V for the LOW-state and 4V to 5.3V for the HIGH-state. The transmitter must be able to drive up to a few mA. There is no limit on transmission speed for an SPI interface. While some high-speed devices operate up to 100MHz (e.g. SD cards), many devices operate up to 10MHz, which is what the I/O Master hardware will target. [11]

SWD is a protocol specifically used for programming and debugging embedded systems. It is a serial interface which operates similarly to I^2C, as it uses two pins: one for bidirectional data (SWDIO) and one for a clock signal (SWCLK). The SWD electrical interface is a subset of the JTAG interface, defined in IEEE 1149.1 (JTAG Test Access Port). Serial wire debug operates at the same level as the supply voltage of the embedded system, typically 3.3V or 5V. Due to its similarity to I^2C, its electrical requirements are identical.

RS-485 is a protocol used for serial communication, similar to RS-232, however the protocol uses differential signalling for the data lines. One differential pair is used for the physical interface, which consists of a wire for the non-inverted signal and the inverted signal. A 120Ω

termination resistor must be placed between the two differential signal lines. Devices measure the voltage difference between the two lines in order to determine the digital state, which allows the interface to be immune to common-mode noise (since the noise is subtracted out), and allows EMI emissions to be minimized (since current flow is opposite between wires, cancelling any magnetic field). The differential pair is bidirectional. The transmission voltage levels are a differential voltage of 0.2V to 6V for the LOW-state and a differential voltage of -6V to -0.2V for the HIGH-state. Common mode voltages on each signal line can be between -7V and 12V. Similar to RS-232, long cables may require high drive currents. Communication speeds can be up to 10Mbit/s. [13]

Table 2: Summary of Protocol Electrical Requirements

	RS-232	**I2C**	**SPI**	**SWD**	**RS-485**
Output Voltage (LOW-state)	-5V to -15V	0V to 0.4V	0V to 0.5V	0V to 0.4V	0.2V to 6V (differential)
Output Voltage (HIGH-state)	5V to 15V	3.5V to 5.5V	4.5V to 5V	3.5V to 5.5V	-6V to -0.2V (differential)
Input Voltage (LOW-state)	-3V to -15V	-0.5V to 1.5V	-0.3V to 1V	-0.5V to 1.5V	0.2V to 6V (differential)
Input Voltage (HIGH-state)	3V to 15V	3.5V to 5.5V	4V to 5.3V	3.5V to 5.5V	-6V to -0.2V (differential)
Output Current	Up to 45mA	Up to 20mA	Up to 5mA	Up to 20mA	Up to 45mA
Frequency	Up to 2MHz	Up to 5MHz	Up to 10MHz	Up to 5MHz	Up to 10MHz
Resistors		- Pull-up on SCL - Pull-up on SDA		- Pull-up SWCLK - Pull-up SWDIO	- Termination between lines

Based on the analysis of the target protocols, the I/O Master hardware must be able to support a wide range of voltages and be able to source or sink a decent amount of current. When driving an output signal, the hardware must have a voltage range of at least -15 to 15V and an output

current of 45mA to satisfy all protocols (RS-232 specifically). In order to ensure a hardware design that is general purpose and is capable of supporting additional protocols in the future, a wider operation range should be selected than necessary with the current set of protocols. The tradeoff however is that this will increase the implementation complexity. For this project, the I/O Master will support a voltage range of -15V to +15V and be able to source or sink up to 50mA.

2.3.2 Single-Ended Communication

A significant portion of the hardware design involves creating a general purpose transmitter and receiver in order to communicate with the target device at its voltage levels. Due to the high voltage, high current, and high speed requirements outlined in the previous section, an off-the-shelf solution in the form of an IC chip is likely not available or only available at high cost. The voltage range is the toughest requirement to meet, and this is illustrated by the fact that for lower voltage applications, there are quite a few high-speed line drivers, transceivers, and buffers that would make this project trivial. As a result, the I/O Master hardware design will likely consist mostly of discrete components.

For single-ended communication, both the low-voltage signals of the I/O Master and high-voltage signals of the target device are ground referenced. As a digital signal, only two voltage states are possible. For low-voltage signals, these states are 0V and 3.3V. For high-voltage signals, the voltage states depend on the protocol in use and the specific operating conditions of the target device. These high-voltage states will be referred to as V_L, the logic-LOW voltage state, and V_H, the logic-HIGH voltage state. A block diagram for a general

purpose transmitter/receiver is shown in Figure 2. This circuit is known as a bidirectional level shifter.

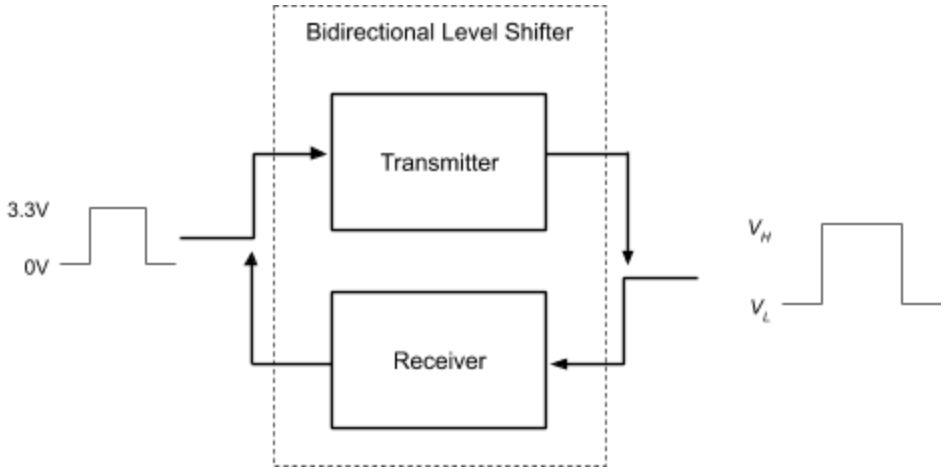


Figure 2: Block Diagram of a General-Purpose Bidirectional Level Shifter

The goal of a general purpose transmitter is to take a low-voltage, low-power signal and generate a high-voltage, high-power signal that can be received by the target device. To achieve this, the low-voltage signal states are correlated with the high-voltage signal states V_L and V_H, and the result is fed through a power amplifier. A block diagram is shown in Figure 3. Since the V_L and V_H logic levels must change when the user selects a different protocol to use, these voltages can be supplied with two variable voltage sources.

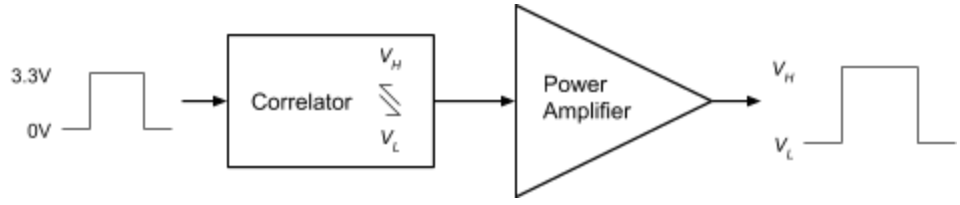


Figure 3: Block Diagram of a General-Purpose Transmitter

This system can be realized with a transistor push-pull amplifier where V_L and V_H form the power rails of the amplifier. The push-pull amplifier combines both the correlator and power amplifier. A simplified circuit diagram is shown in Figure 4.

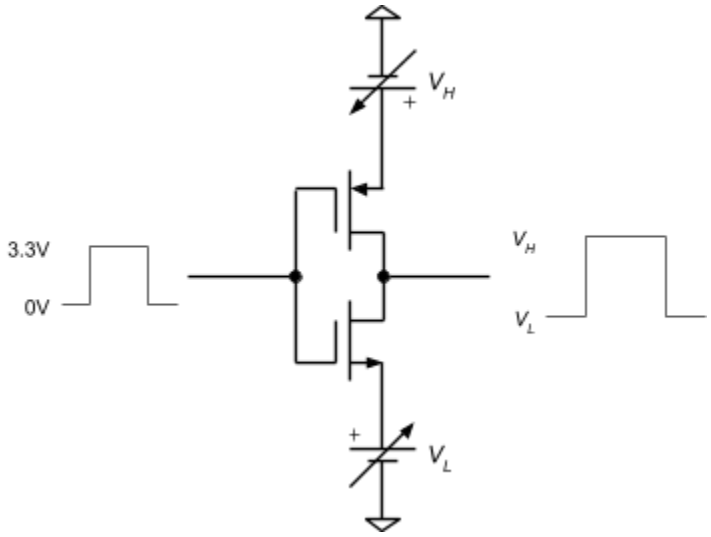


Figure 4: Push-Pull Amplifier as a General-Purpose Transmitter

A general purpose receiver is less straightforward. The receiver must take a high-voltage signal from the target device and generate a low-voltage signal that can be received by the I/O Master. Correlation of the high-voltage signal states to the low-voltage signal states is more difficult than with the transmitter because for each high-voltage state, a range of voltages are valid. The simplest solution would be to compare the high-voltage signal to a threshold voltage to determine the low-voltage signal state. Assuming that the incoming high-voltage signal has states that, on average, are close to V_H and V_L, then the optimal threshold would be the average of V_H and V_L:

$$V_{TH} = \frac{V_H + V_L}{2}$$

A block diagram of such a receiver is shown in Figure 5.

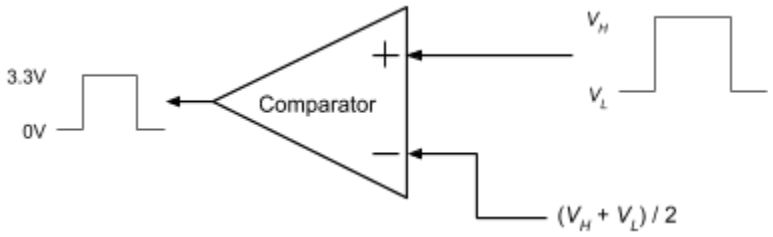


Figure 5: Block Diagram of a General-Purpose Receiver

2.3.3 Differential Communication

The ability to transmit and receiver differential signals greatly expands the applicability of the I/O Master compared to other currently available devices. With differential signalling, two lines are used for one signal, where the signal is transmitted as the voltage difference between the two lines as opposed to referenced to ground. This type of signalling is useful for high-speed communications because it reduces the effect of common mode noise on the line and it reduces the EMI interference that the line causes on other devices. Since the voltage of the two lines is subtracted to obtain the signal, common mode noise that affects both lines equally is subtracted out and has no effect. If the voltage levels of the two lines are chosen such that the current direction in one line is opposite the direction of the other line, then the magnetic field generated by one line is cancelled out by the magnetic field of the other line, resulting in much lower EMI emissions.

As with single-ended communication, a differential level shifter consists of a transmitter and a receiver. In this case however, the level shifter must also convert between single-ended and differential signalling schemes.

2.4 Software Design

2.4.1 Microprocessor Functionality

There are some key features in a microprocessor that are utilized for the I/O Master. The first is the utilization of direct memory access (DMA). DMA is used to stream input and output data and is further explained in Section 2.4.3. Onboard timers are also utilized for the I/O Master. The option to use an external clock signal to count a timer is used in conjunction with a clock signal of a target device. To interface with a computer, the microcontroller needs to utilize on-board USB hardware, which helps offload the burden from the CPU of sending and receiving data. USB is explained more in Section 2.4.2. DACs are needed to control logic levels for the target device. Due to the high number of DACs needed for the project, it is necessary to use an external I2C controlled DAC. Device configuration is further explained in Section 2.4.5. (CS/AD)

2.4.2 USB Protocol

The I/O Master utilizes USB for communication with the user's computer. The USB protocol is very broad and has many different standards, so more information is necessary to determine how USB can be used for communication. The STM32H7 microcontroller features built-in hardware support for USB 1.1 Full Speed that has a theoretical maximum throughput of 12 Mbps. With the addition of some external hardware, the microcontroller can also operate at USB 2.0 High Speed with a theoretical maximum of 480 Mbps.

Since the USB 1.1 throughput is 12 Mbps, it potentially could be used, though the theoretical maximum is close to the necessary speed of 10 Mbps. To determine if using USB 1.1 is possible, the configurations and corresponding overheads needs to be considered. The best suited USB 1.1

device class for the I/O Master is the Communications Design Class (CDC), which is for sending and receiving generic data.

The STM32 CDC library supports a max data packet size of 64 bytes. In addition, the packet can be sent with a top speed of 1 packet / ms, or 1000 packets / second. This equals out to a maximum data throughput of 500 Kbit, which is 20 times slower than the 10 Mbit data throughput required.

USB 2.0 High Speed allows for up to 480 Mbit transfer speeds and offers more device classes to configure. Many high speed and low latency applications utilize the USB 2.0 CDC Network Control Model (NCM) specification. Real world maximums in bandwidth for USB CDC NCM is around 200 Mbit, which is more than enough to be used for 10 Mbit data transfer. USB CDC NCM is very fast because it implements the IEEE 802.3 network frame definitions which now support gigabit speeds over ethernet.

By using USB CDC NCM, all operating systems will already have native high speed support for communication since it is the same network stack that is used by ethernet. The I/O Master is seen as a custom network device, and is easy to interface with it through common libraries in computer software. The real bottleneck of such an implementation is the speed of the USB signal, which can be up to 480 MHz, much higher than the 10 MHz speed. If the overhead of each packet is grossly overestimated as high as 50%, speed of over 100 Mbit is still achievable, thus making USB transfer feasible for the I/O Master as a communications method to the computer. (CS)

2.4.3 DMA

One common method to output high speed data on GPIO pins is using bit banging. The concept of bit banging is counting CPU clocks to time reading and writing of data out to GPIO. However, bit banging requires a significant amount of CPU resources and is difficult to time correctly. Utilizing bit banging requires a complex scheduling algorithm to ensure that timing for inputting and outputting data is exact. With the desire to design the I/O Master to support a varying frequency up to 10 MHz, it is very difficult to achieve a general purpose implementation with bit banging.

Rather than bit banging, DMA can be used to stream data to GPIO pins. With peripheral DMA, the DMA can be clocked and configured to send a block of data every time a timer period finishes.

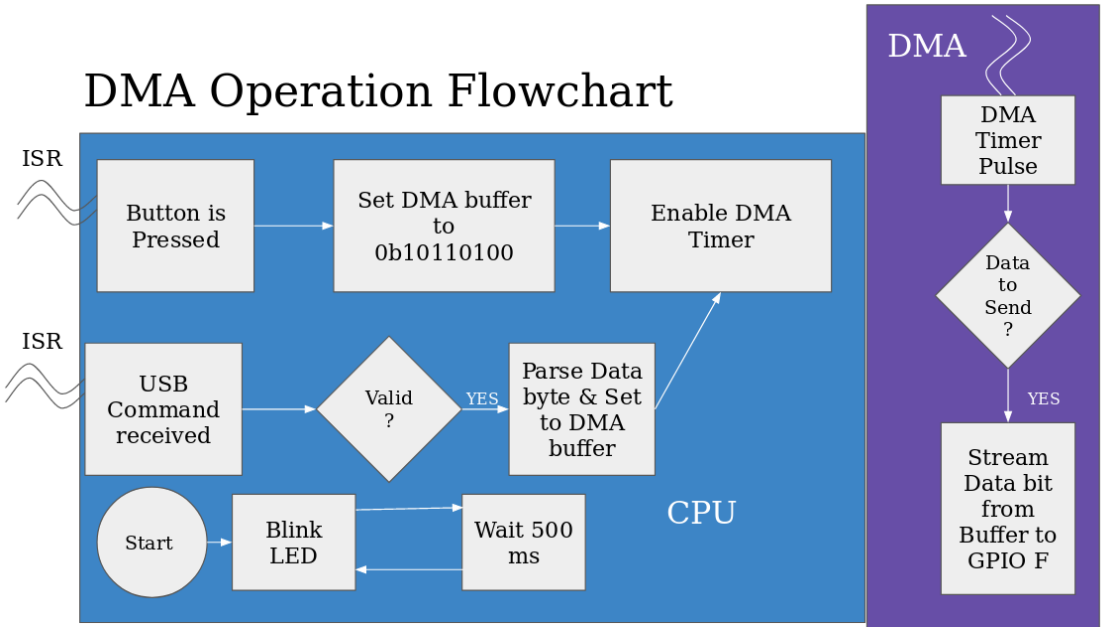


Figure 6: DMA Operation Example Flowchart

In Figure 6, an example flowchart of streaming data using DMA is shown. The CPU is responsible for setting up the DMA buffer and enabling DMA, shown in the blue region. The DMA has its own clock controlled from a timer to send data at every pulse which is shown in the purple region. While data is being streamed, more data can be prepared by the CPU to be streamed since the CPU is not locked up. If bit banging is used, the CPU is responsible for preparing the data and outputting it, making it difficult to output data at a configured frequency accurately and avoid lock ups.

There is also potential to use DMA to move data in and out of USB buffers. Since DMA does not utilize CPU resources, it is most ideal to use DMA in as many situations as possible. Using DMA for a USB buffer does not require a link to a timer clock as the goal for USB is to move data in and out as fast as possible. Hardware USB already has its own clocks to input and output the data at the protocol specification frequency. (CS)

2.4.4 Scheduler

In order to ensure that transfer speeds can handle the 10 MHz requirements of the I/O Master, a scheduler is necessary to keep watch on how full the input and output buffers are, handle notifications of new data, and ensure that data is outputted in a reasonable time.

The microcontroller only has a single core processor, but data could be coming in or moving out of multiple sources at the same time. As a result, a scheduler is necessary in order to effectively accomplish the goal of moving data at 10 MHz speed, which is necessary to support RS-485. The

CPU is responsible for enforcing the schedule. Other onboard devices that do not utilize CPU resources (DMA, Timers, Hardware USB) handle the actual transmission of the data.

As data from the target device moves into the input data buffer on the microcontroller, the scheduler ensures that the data is formed into packets and sent over USB to the computer before the buffer fills up. At USB High Speed specification, data can theoretically be sent at up to 480 MHz. However, there is some overhead to sending a packet, so it would not be ideal to send every frame of data as an individual packet. In addition, sending the USB packet to a computer is a lower priority than reading in the input data. As a result, the input buffer should be configured to be larger than the amount of data that can fit in one USB packet, and the packet should only be output once that buffer is halfway full or if reading data is halted by the user.

Since the microcontroller will be significantly faster than the incoming data rate, the packet can be sent before the buffer is full again. This method ensures that locks on data processing are not needed, which makes it easy to avoid potential data loss through overflow.

In the other direction, data comes from the computer over USB and is then sent to the target device. Since the USB protocol and computer are significantly faster than the max output frequency of 10 MHz, there is a danger of running out of memory through overflow of USB packets. To prevent this, the computer must limit the rate of data sent to the microcontroller.

To prevent an overflow of data coming from the computer, the microcontroller can respond with an *OK* packet if the data is accepted into the buffer. If the buffer is full, then the microcontroller can hold the packet until it can be processed. The computer does not send any additional packets

until the *OK* packet is received. If the microcontroller receives additional data packets, they can be ignored. (CS)

2.4.5 Configuration

To easily switch between protocols, the I/O Master's microcontroller is able to change the hardware's operating parameters based on commands via USB. The microcontroller will be able to configure the clock speed, designate input and output pins, and set the logic voltage levels to match the command specifications.

To control the transmission frequency for data, the system clock and timers available will be reconfigured to match the desired frequency as closely as possible. The frequency created will be reported back to the user through the GUI at which point they will determine if it suits their needs.

The Level Shifter includes multiple DACs to generate the logic-high voltage, V_H, and the logic-low voltage, V_L. These two voltages are used to output a signal to the target device, and also used to determine the voltage threshold when receiving a signal from the target device. Since the DAC itself can only output 0 to 3.3V, each output has an op-amp buffer circuit which scales the voltage to the range of -15 to 15V. The DACs are controlled using I^2C commands from the microcontroller. (AD)

2.4.6 GUI Language

The GUI is written entirely in Python. More specifically, it is written in tkinter. Tkinter was chosen mainly because it is open source and it also comes standard with the Python package.

Because of these things, it has a large online community that uses it. This means it is easy to get help online for any issues that may occur while using it and documentation on how to use it is easy to find and read. This provides many different ways to solve the same problem and allows for the best solution to be chosen or for new ideas to be generated off of the found solutions. Tkinter also has an easy to use packing system for the layout of the interface because the pack command will layout everything on its own thus avoiding the need to assign everything down to the pixel for laying things out. On the other hand it has the option to use a grid system which allows for more control over how things look. Creating all the widgets is simple and straightforward as well. Built in classes along with the option to build more advanced classes with the built in one allows for easy to use versatile widgets. Python also allows for multithreading which makes receiving data possible. This makes it possible to use the GUI for sending data while the GUI is looking for recieved data in the background. (CD)

3. Engineering Requirements Specification

Table 3: I/O Master Engineering Requirements

Marketing Requirement	Engineering Requirement	Justification
1, 6	The I/O Master shall be powered from either a standard USB port (5V DC, 500mA) or external DC source (12V DC, 500mA).	User convenience, and allows an external supply if more power is needed.
2, 5	The I/O Master shall be capable of transmitting and receiving digital signals between -15V and +15V DC on each I/O pin.	Required to support all of the target protocols.
2	The I/O Master shall be capable of sourcing and sinking up to 50mA on each I/O pin.	Required to support all of the target protocols.
2, 5	The I/O Master shall have a minimum of 4 fully-configurable I/O pins.	Required to support all of the target protocols, specifically SPI.
6	The I/O Master shall have dedicated power and ground pins for powering the target device, selectable between 3.3V and 15V DC, and provide up to 100mA.	User convenience.
1	The I/O Master shall have visual indicators of device readiness and communications activity.	Provide feedback to the user to ensure the device is easy to use.
4	The I/O Master shall be able to withstand over-voltages up to -30V to +30V DC and limit over-currents to 50mA on each I/O pin.	Tolerate user error to ensure that the device is easy to use and safe.
1, 2, 3	The I/O Master shall be able to transmit and receive data using I^2C, SPI, RS-232, SWD, and RS-485 protocols.	Support common protocols and ensure that the hardware can support a wide range of electrical requirements.
2	The I/O Master shall be capable of transmitting and receiving digital signals up to 10MHz.	Support the high speed operation of target protocols, and to improve upon the Bus Pirate's 1MHz communication limit.
1	The I/O Master shall support USB 2.0 high speed communication to a computer.	Communicate with a computer at a rate at least as fast at target protocol speed (10MHz digital signal → up to 10Mbit/s theoretical)
2	The I/O Master shall be capable of transmitting and receiving differential digital signals.	Required to support all of the target protocols, specifically RS-485.

4. Engineering Standards Specification

Table 4: I/O Master Engineering Standards

	Standard	**Use**
Safety	N/A	
Communication	USB UART SPI I2C SWD RS-485	USB for communication with a computer. I2C for communication with on-board external DAC, UART, SPI, I2C, SWD, RS-485 for I/O Master design to communicate with target devices
Data Formats	USB CDC NCM	Data format for communication between computer and I/O Master
Design Methods	Block Diagrams, UML Sequence Diagrams, Flow charts	
Programming Languages	C, Python	Microcontroller Firmware, Computer Software
Connector Standards	USB	Communication with computer and I/O Master

5. Accepted Technical Design

5.1 Hardware Design

5.1.1 General Overview

The I/O Master hardware consists of three main subsystems and three voltage regulators. Figure 7 shows a level 1 block diagram of the system and the various connections between the subsystems. The hardware can be broken down into two separate signal domains. So-called “low voltage” signals are 3.3V-level signals and include the computer interface and microcontroller subsystem. So-called “high voltage” signals are at the voltage level of the target device, which depending on the signal may be anywhere from -15V to +15V and include the signals that pass through the circuit protection subsystem to the target device. As the I/O Master consists of multiple I/O pins, the level shifter and circuit protection subsystems are repeated for each I/O pin, giving each individual pin full configurability and circuit protection independent from other pins. There is one level shifter subsystem for each pair of I/O pins, and there is one circuit protection subsystem for each I/O pin.

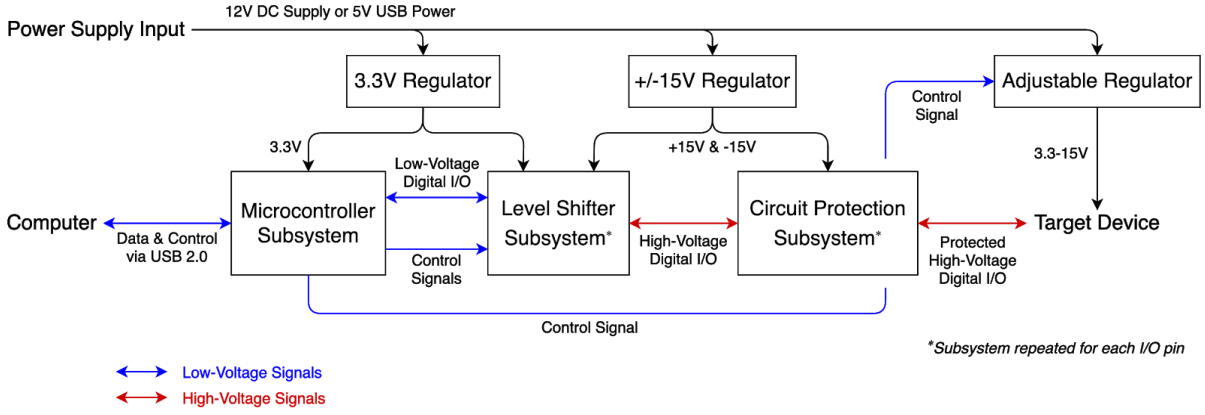


Figure 7: Hardware Level 1 Block Diagram

The microcontroller subsystem has two tasks: configuring the other hardware subsystems and transferring data between the digital interface and the computer. Depending on the protocol selected, the level shifter subsystem must be configured for the electrical requirements of the protocol, including logic voltage levels, pull-up/pull-down resistors, and single-ended/differential communication modes. The adjustable regulator must also be configured for the correct voltage to power the target device. The microcontroller takes configuration commands from the computer and outputs the correct signals, whether a digital signal or analog voltage, to configure the hardware. Once configured, the microcontroller is responsible for taking digital data from the computer and generating the proper digital signal for the protocol selected as input to the level shifter. Conversely, the microcontroller must read received signals from the level shifter and send the received digital data to the computer.

The level shifter subsystem separates these two signal domains and shifts signals from the low-voltage domain to the high-voltage domain and vice versa. Based on a configuration signal from the microcontroller, the level shifter generates two logic voltage rails, V_H and V_L, used for high-voltage signalling. Since a low-voltage signal state can be either 0V or 3.3V, the corresponding high-voltage signal state is either V_L or V_H, respectively. The logic voltage rails are also used to generate a threshold voltage for interpreting the state of received signals. As all microcontroller signals are single-ended, the level shifter subsystem is also responsible for converting single-ended signals to differential signals and vice versa. For this reason, there is one level shifter subsystem for each pair of I/O pins. In differential mode, I/O pins are paired together and two pairs of logic voltage rails, V_{H1},V_{L1} and V_{H2},V_{L2}, are used to generate differential voltages.

The circuit protection subsystem is designed to protect the target device from over-current and over-voltage related failures. To achieve this, the voltage is limited using zener diodes to clamp the voltage while the over-current protection is made of op-amps, an NMOS transistor and a voltage divider to control the gate voltage.

To power the various subsystems, there are three separate voltage regulators. The 3.3V regulator generates a 3.3V voltage rail for powering the microcontroller subsystem and low-voltage side of the level shifter subsystem. The ±15V regulator generates two opposite voltage rails, +15V and -15V, used for powering the high-voltage side of the level shifter subsystem and circuit protection subsystem. Finally, the adjustable regulator is used solely for powering the target device, as long as the target device's power requirements are small. A control signal from the microcontroller can set the output voltage anywhere between 3.3V and 15V.

Tables 5, 6, 7, 8, 9, and 10 list the functional requirements for the subsystems shown in Figure 7.

Table 5: Microcontroller Subsystem Functional Requirements

Module	Microcontroller Subsystem
Designer	Ian Glen & Nik Untch
Inputs	- 3.3V power input - USB interface - Low-voltage signals from level shifter subsystem
Outputs	- USB interface - Low-voltage signals to level shifter subsystem - Control signals to level shifter subsystem - Control signal to adjustable regulator
Description	The microcontroller is responsible for configuring the level shifter hardware for the protocol that the user has selected and transferring data between the USB interface and low-voltage digital I/O.

Table 6: Level Shifter Subsystem Functional Requirements

Module	Level Shifter Subsystem
Designer	Ian Glen
Inputs	- 3.3V power input - ±15V power input - Low-voltage signals from microcontroller subsystem - Control signals from microcontroller subsystem - High-voltage signals from circuit protection subsystem
Outputs	- Low-voltage signals to microcontroller subsystem - High-voltage signals to circuit protection subsystem
Description	The I/O level shifter is responsible for translating digital signals between the low voltages required by the microcontroller and the high voltages required by the target device. For protocols that require differential signalling, this module also converts the high voltage signals to single-ended signalling as required by the microcontroller. This module accepts control signals from the microcontroller so that its hardware can be configured in specific ways depending on the protocol selected.

Table 7: Circuit Protection Subsystem Functional Requirements

Module	Circuit Protection Subsystem
Designer	Nik Untch
Inputs	- ±15V power input - High-voltage signals from level shifter subsystem - High-voltage signals from target device
Outputs	- High-voltage signals to level shifter subsystem - High-voltage signals to target device
Description	The I/O circuit protection module protects the Target Device against issues such as overcurrent and overvoltage, and protects the I/O Master from reverse polarity

Table 8: 3.3V Regulator Functional Requirements

Module	3.3V Regulator
Designer	Nik Untch
Inputs	- 5V to 12V power input
Outputs	- Regulated 3.3V power rail
Description	The 3.3V regulator is designed to provide a 3.3V power rail for the microcontroller subsystem and low-voltage side of the level shifter subsystem.

Table 9: ±15V Regulator Functional Requirements

Module	±15V Regulator
Designer	Ian Glen
Inputs	- 5V to 12V power input
Outputs	- Regulated +15V power rail - Regulated -15V power rail
Description	The ±15V is designed to provide both +15V and -15V power rails for the high-voltage side of the level shifter subsystem and the circuit protection subsystem.

Table 10: Adjustable Regulator Functional Requirements

Module	Adjustable Regulator
Designer	Nik Untch
Inputs	- 5V to 12V power input - Control signal from microcontroller subsystem
Outputs	- Regulated power rail, adjustable between 3.3V to 15V
Description	The adjustable regulator is designed to provide a regulated power rail for the purpose of powering the target device. Depending on the control signal, the output voltage is adjustable between 3.3V and 15V.

5.1.2 Microcontroller Overview

The microcontroller subsystem consists of the STM32H7 microcontroller and required components to support and interface with the microcontroller. A level 2 block diagram of the microcontroller subsystem is shown in Figure 8.

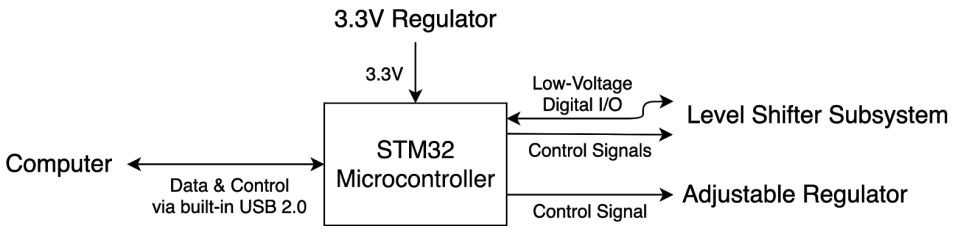


Figure 8: Microcontroller Subsystem Level 2 Block Diagram

The STM32H7 is powered by the 3.3V regulator. Per the datasheet, additional passive components provide decoupling and filtering on the microcontroller power input pins. Since the microcontroller includes a built-in USB peripheral and PHY, the USB connection from a computer is wired directly to the microcontroller. Using the microcontroller's built-in digital to analog converter (DAC) and various I/O pins, control signals can be generated to configure the other hardware of the I/O Master. Given the correct clock frequency for the protocol and digital data from the computer, the microcontroller can generate a digital signal on its I/O pins to serve as input to the level shifter. This is accomplished using the microcontroller's integrated direct memory access (DMA) controller and various software techniques to achieve high speed communication, which are described in detail in the software design section. The same techniques are used when receiving a digital signal from the level shifter. (IG)

Table 11 lists the functional requirements for the block shown in Figure 8.

Table 11: STM32 Microcontroller Functional Requirements

Module	STM32 Microcontroller
Designer	Ian Glen & Nik Untch
Inputs	- 3.3V power input - USB commands/data from computer - Low-voltage digital signals from level shifter subsystem - Clock input from VCO (1kHz to 10MHz)
Outputs	- USB data to computer - Low-voltage digital signals to level shifter subsystem - Control signal to VCO (analog voltage, 0-3.3V) - Control signals to other subsystems
Description	The STM32 microcontroller is responsible for configuring the other subsystems and for transferring data between the computer and level shifter.

5.1.3 Level Shifter Overview

The level shifter subsystem consists of a logic level generator, output driver, single-ended receiver, differential receiver, and various configurable resistors. A level 2 block diagram of the level shifter subsystem is shown in Figure 9. For each I/O pin, there exists one output driver, one single-ended receiver, and various configurable resistors. As some protocols use a single I/O line for bidirectional communication, the output of the output driver is tied to the input of the single-ended receiver. When bidirectional communication is required, the output driver can be put into a floating state in order to make receiving possible. For each adjacent pair of I/O pins, there is a differential receiver and a multiplexer to switch between single-ended input or differential input to the microcontroller. In differential mode, only the multiplexed input from the differential receiver is read by the microcontroller, while the other input from the second single-ended receiver is ignored. Each I/O pin has two logic levels, a low-level voltage V_L and a high-level voltage V_H, which are determined by the logic level generator.

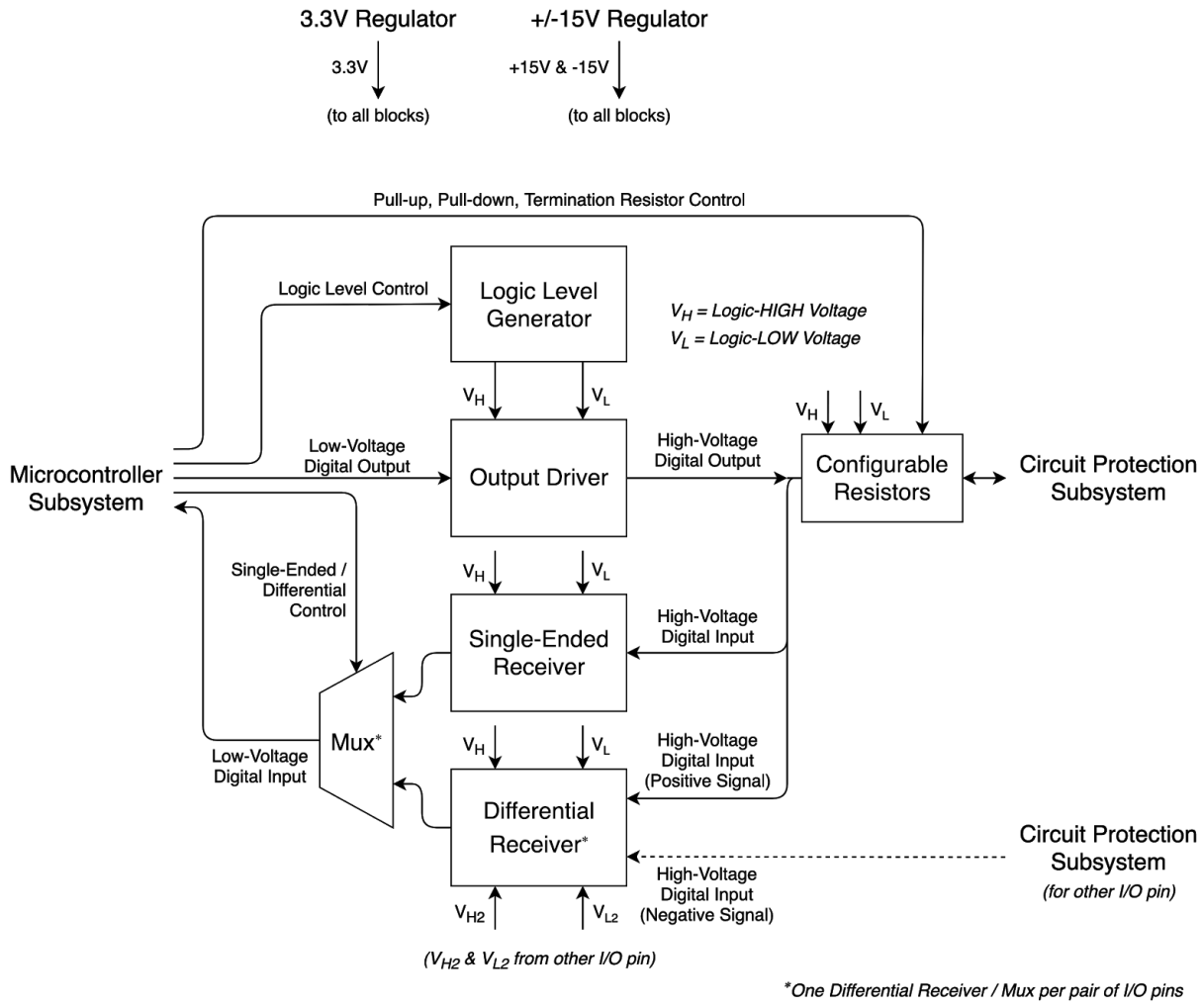


Figure 9: Level Shifter Subsystem Level 2 Block Diagram

Tables 12, 13, 14, 15, 16, and 17 list functional requirements for the blocks shown in Figure 9.

Table 12: Logic Level Generator Functional Requirements

Module	Logic Level Generator
Designer	Ian Glen
Inputs	- 3.3V power input - ±15V power input - Control signal from microcontroller subsystem (I^2C bus)
Outputs	- V_H voltage rail for each I/O pin (configurable between -15V and +15V) - V_L voltage rail for each I/O pin (configurable between -15V and +15V)
Description	The logic level generator is responsible for generating the logic level voltages for each individual I/O pin, as defined by the microcontroller.

Table 13: Output Driver Functional Requirements

Module	Output Driver
Designer	Ian Glen
Inputs	- 3.3V power input - ±15V power input - Low-voltage signals from microcontroller
Outputs	- High-voltage signal to target device
Description	The output driver is responsible for shifting a low-voltage digital signal from the microcontroller to the high-voltage logic levels required by the target device, as well as acting as a power amplifier to provide the required drive current.

Table 14: Single-Ended Receiver Functional Requirements

Module	Single-Ended Receiver
Designer	Ian Glen
Inputs	- 3.3V power input - ±15V power input - High-voltage signal from target device
Outputs	- Low-voltage signal to microcontroller
Description	The single-ended receiver is responsible for shifting a high-voltage signal from the target device to a low-voltage signal so that it can be received by the microcontroller.

Table 15: Differential Receiver Functional Requirements

Module	Differential Receiver
Designer	Ian Glen
Inputs	- 3.3V power input - ±15V power input - High-voltage signal from target device (first I/O pin - non-inverted signal) - High-voltage signal from target device (second I/O pin - inverted signal)
Outputs	- Low-voltage single-ended signal to microcontroller
Description	The differential receiver is responsible for shifting a high-voltage differential signal from the target device to a low-voltage single-ended signal so that it can be received by the microcontroller.

Table 16: Differential Multiplexer Functional Requirements

Module	Differential Multiplexer
Designer	Ian Glen
Inputs	- 3.3V power input - Control signal from microcontroller
Outputs	- Low-voltage signal from single-ended receiver - Low-voltage signal from differential receiver
Description	The multiplexer is responsible for selecting between the single-ended receiver input or differential receiver input depending on the operating mode of the I/O pin.

Table 17: Configurable Resistors Functional Requirements

Module	Configurable Resistors
Designer	Ian Glen
Inputs	- 3.3V power input - ±15V power input - Control signals from microcontroller
Outputs	n/a
Description	The configurable resistors are responsible for properly biasing the I/O pins depending on the electrical requirements of the particular protocol in use.

The logic level generator uses a combination of a multi-channel DAC and buffer amplifier to generate low current voltage rails for use as logic levels in high-voltage signalling. A level 3

block diagram is shown in Figure 10. The DAC is a Microchip MCP4728, which provides four 12-bit channels. Each I/O pin uses two DAC channels to generate the V_L and V_H logic level voltages. Via I^2C, the microcontroller sets the DAC channel output voltages, which can be anywhere between 0V and 3.3V. Typically DACs must be buffered in order to prevent a load from affecting the output voltage. A combination of buffer and scaler is used to scale the output voltage to -15V to +15V, allowing for a full range of logic levels to be selected. Given that each I/O pin is limited to 50mA and each DAC channel is only tied to one I/O pin, a simple buffer/scaler circuit is capable of providing enough current. (IG)

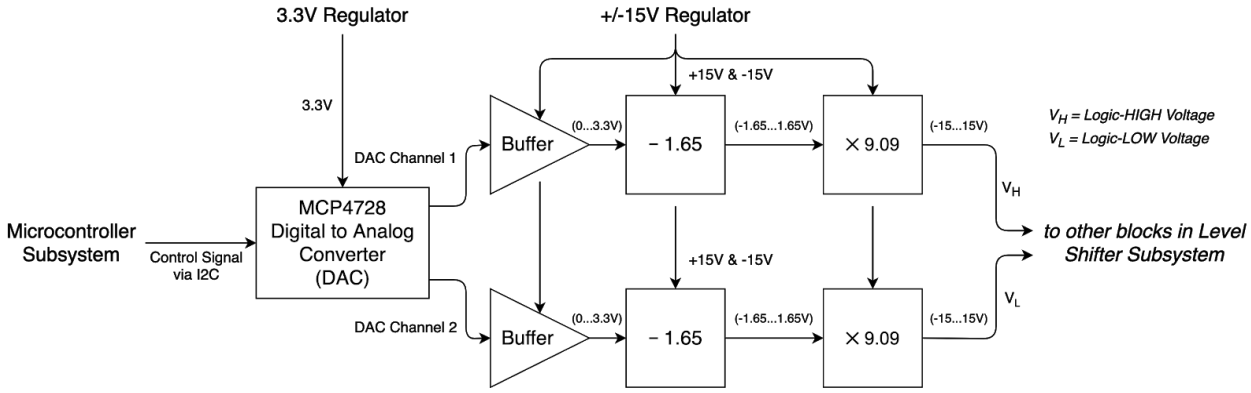


Figure 10: Logic Level Generator Level 3 Block Diagram

The DAC register value can be calculated for a given output voltage with the following equation:

$$DAC\_VALUE = \frac{2^{12} - 1}{30} \times (15 - V_{OUT})$$

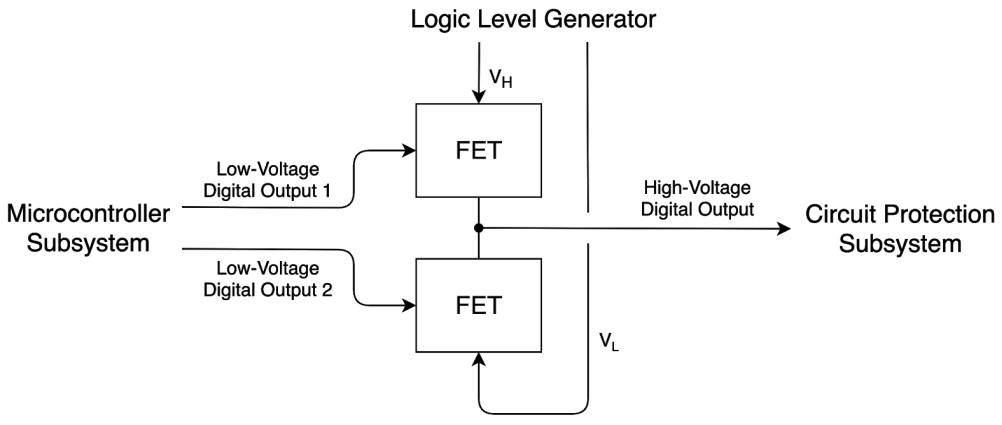
Tables 18 and 19 list the functional requirements for the blocks shown in Figure 10.

Table 18: Digital to Analog Converter (DAC) Functional Requirements

Module	Digital to Analog Converter (DAC)
Designer	Ian Glen
Inputs	- 3.3V power input - Control signal from microcontroller (I^2C bus)
Outputs	- V_H prescaled voltage (0V to 3.3V) - V_L prescaled voltage (0V to 3.3V)
Description	The DAC is responsible for providing configurable voltages that will be scaled to generate the logic level voltages.

Table 19: DAC Buffer/Scaler Functional Requirements

Module	DAC Buffer/Scaler
Designer	Ian Glen
Inputs	- 3.3V power input - ±V power input - Prescale voltages from DAC
Outputs	- V_H voltage rail for each I/O pin (configurable between -15V and +15V) - V_L voltage rail for each I/O pin (configurable between -15V and +15V)
Description	The DAC buffer/scaler is responsible for buffering the DAC output voltages and scaling them to create the logic level voltages.



LV Output 1	LV Output 2	HV Output
LOW	LOW	High-Z
HIGH	LOW	HIGH
LOW	HIGH	LOW
HIGH	HIGH	Not Possible

Figure 11: Output Driver Level 3 Block Diagram

Table 20 lists the functional requirements for the FET block shown in Figure 11.

Table 20: FET Functional Requirements

Module	FET
Designer	Ian Glen
Inputs	- V_H and V_L logic level power input - Low-voltage signal from microcontroller (3.3V, up to 10MHz)
Outputs	- High-voltage signal to target device (up to ±15V, up to 10MHz, up to 50mA)
Description	The FET is responsible for shifting the low-voltage microcontroller signal to a higher voltage and acting as a power amplifier to provide the required drive current.

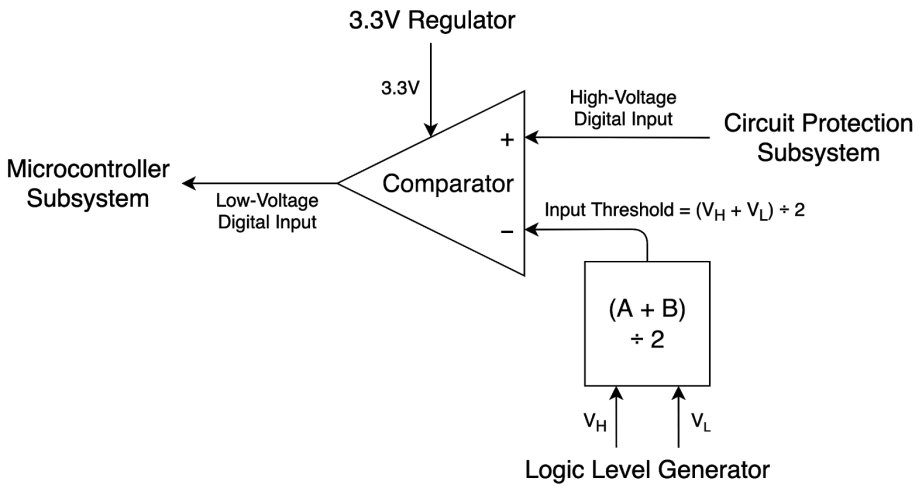


Figure 12: Single-Ended Receiver Level 3 Block Diagram

Tables 21 and 22 list functional requirements for the blocks shown in Figure 12.

Table 21: Receiver Comparator Functional Requirements

Module	Receiver Comparator
Designer	Ian Glen
Inputs	- 3.3V power input - High-voltage signal from target device - Threshold voltage generated from logic level voltage rails
Outputs	- Low-voltage signal to microcontroller
Description	The Comparator is responsible for shifting a high-voltage signal from the target device to a low-voltage signal so that it can be received by the microcontroller.

Table 22: Receiver Threshold Functional Requirements

Module	Receiver Threshold
Designer	Ian Glen
Inputs	- V_L and V_H logic level voltage rails
Outputs	- Receiver threshold voltage
Description	The Receiver Threshold block is responsible for generating a threshold that is the average of the logic level voltage rails used for detection of the signal from the target device.

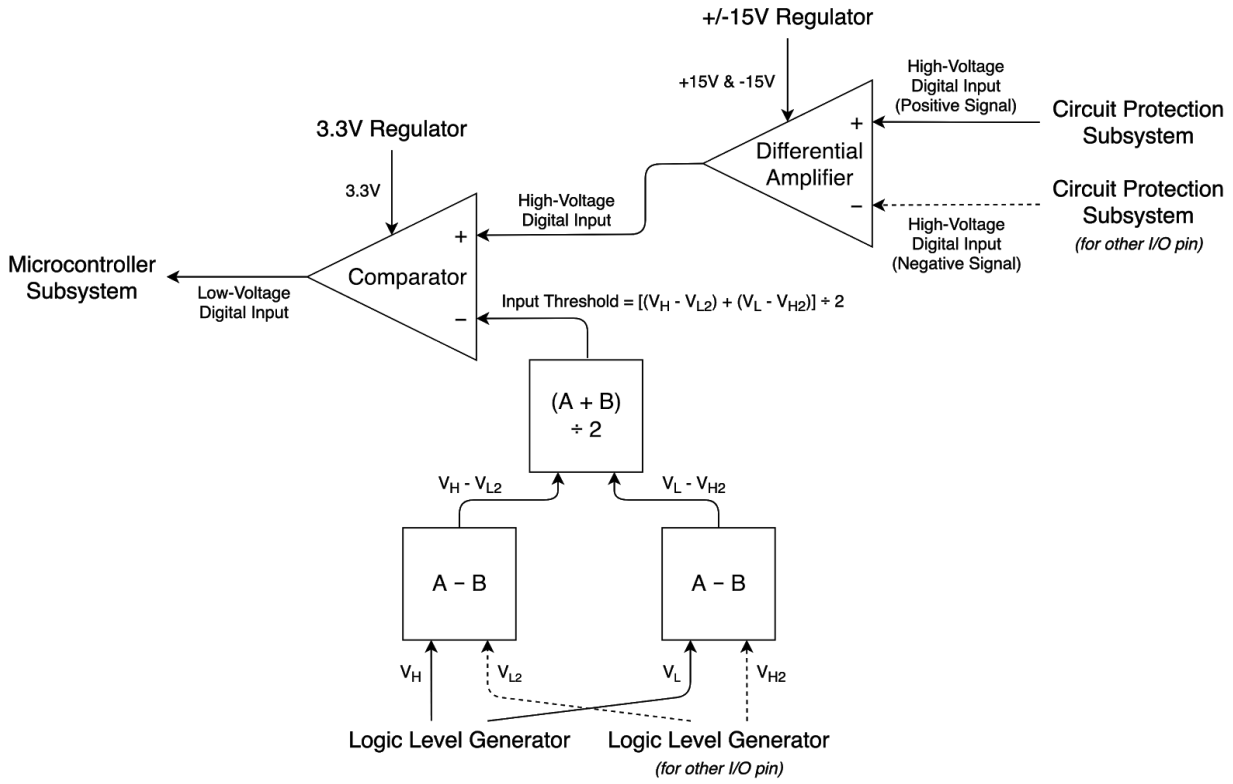


Figure 13: Differential Receiver Level 3 Block Diagram

Tables 23, 24, and 25 list the functional requirements for the blocks shown in Figure 13.

Table 23: Differential Receiver Amplifier Functional Requirements

Module	Differential Receiver Amplifier
Designer	Ian Glen
Inputs	- ±15V power input - High-voltage signal from target device (non-inverted signal) - High-voltage signal from target device (inverted signal)
Outputs	- Single-ended high-voltage signal to receiver comparator
Description	The Differential Receiver Amplifier is responsible for measuring the differential voltage between two I/O pins in order to convert it to a single-ended signal.

Table 24: Differential Receiver Comparator Functional Requirements

Module	Differential Receiver Comparator
Designer	Ian Glen
Inputs	- 3.3V power input - High-voltage single-ended signal from differential amplifier - Threshold voltage generated from logic level voltage rails
Outputs	- Low-voltage signal to microcontroller
Description	The Differential Receiver Comparator is responsible for shifting the high-voltage single-ended signal from the differential amplifier to a low-voltage signal so that it can be received by the microcontroller.

Table 25: Differential Receiver Threshold Functional Requirements

Module	Differential Receiver Threshold
Designer	Ian Glen
Inputs	- V_L and V_H logic level voltage rails from the non-inverted I/O pin - V_L and V_H logic level voltage rails from the inverted I/O pin
Outputs	- Receiver threshold voltage
Description	The Differential Receiver Threshold block is responsible for generating a receiver threshold voltage that is the average of the difference between the logic level voltages for the two I/O pins used for differential communication.

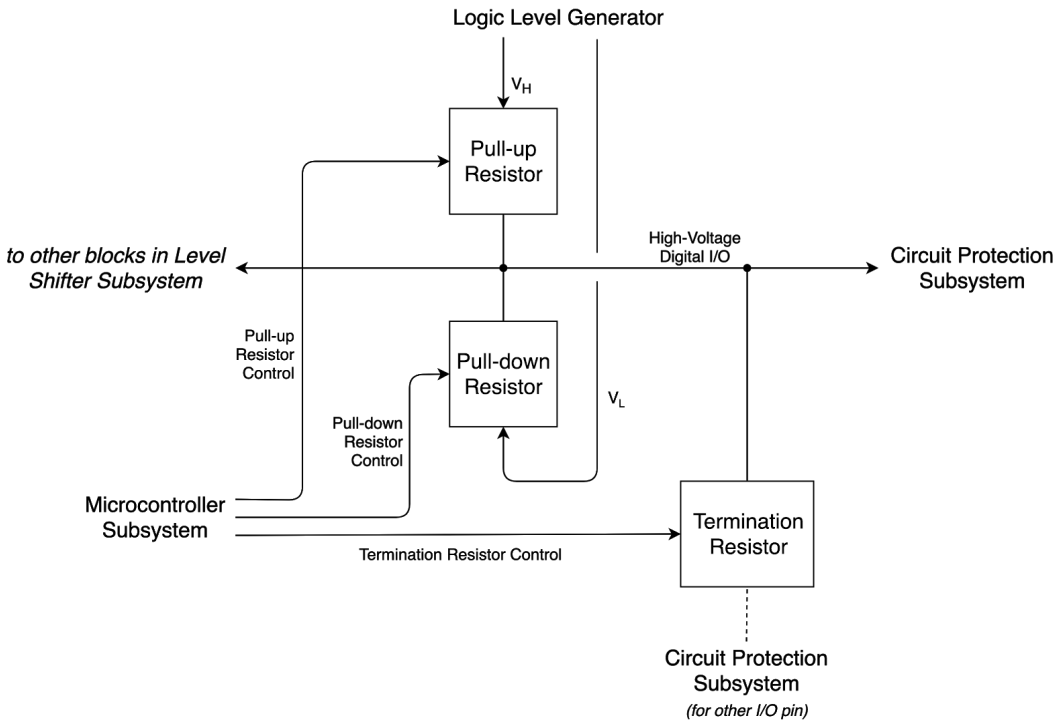


Figure 14: Configurable Resistors Level 3 Block Diagram

Tables 26, 27, and 28 list the functional requirements for the blocks shown in Figure 14.

Table 26: Pull-up Resistor Functional Requirements

Module	Pull-up Resistor
Designer	Ian Glen
Inputs	- High-voltage signals from target device
Outputs	n/a
Description	The Pull-up Resistor block is a resistor that can be enabled by the microcontroller to provide a pull-up resistance if required by the protocol.

Table 27: Pull-down Resistor Functional Requirements

Module	Pull-down Resistor
Designer	Ian Glen
Inputs	- High-voltage signals from target device
Outputs	n/a
Description	The Pull-down Resistor block is a resistor that can be enabled by the microcontroller to provide a pull-down resistance if required by the protocol.

Table 28: Termination Resistor Functional Requirements

Module	Termination Resistor
Designer	Ian Glen
Inputs	- High-voltage signal from target device (non-inverting signal) - High-voltage signal from target device (inverting signal)
Outputs	n/a
Description	The Termination Resistor block is a resistor that can be enabled by the microcontroller to provide the correct termination resistance between differential signal lines if required by the protocol.

5.1.4 Circuit Protection Overview

The level 2 block diagram for the Circuit Protection subsystem is shown in Figure 15. The level 3 block diagrams for the blocks in Figure 15 are shown in Figures 16 and 17.

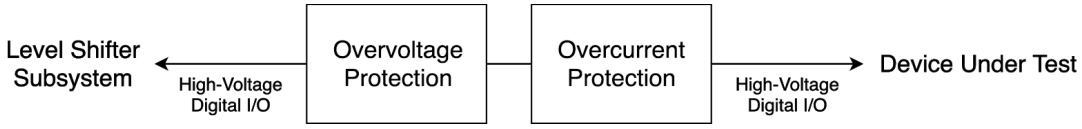


Figure 15: Circuit Protection Subsystem Level 2 Block Diagram

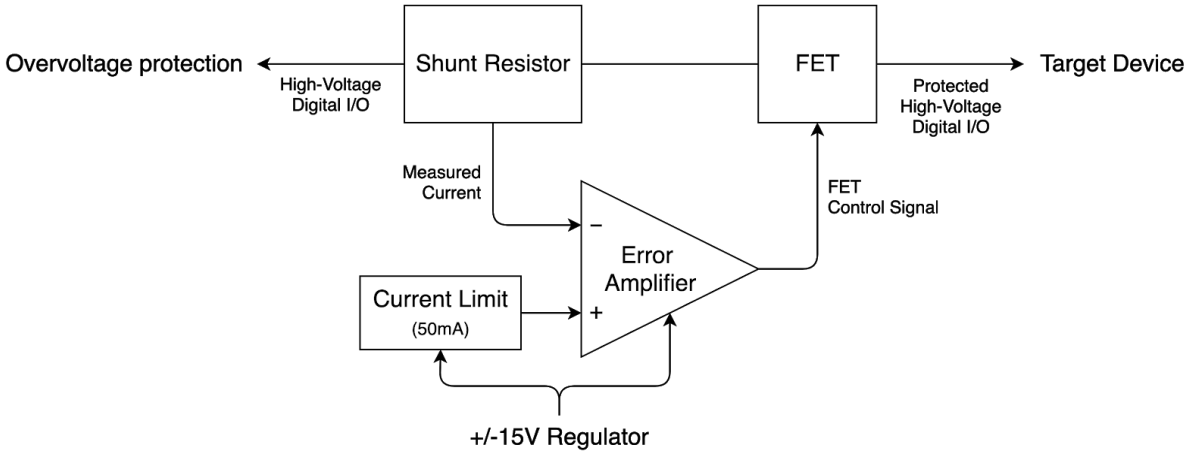


Figure 16: Over-current Protection Level 3 Block Diagram

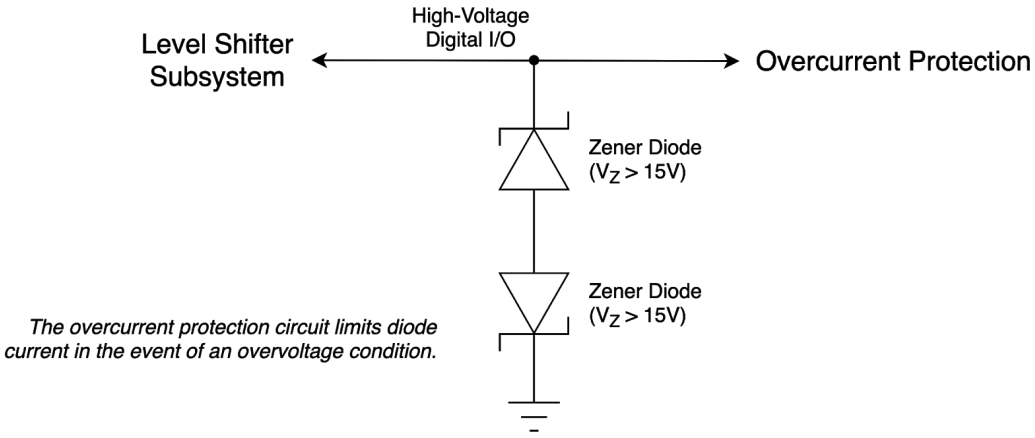


Figure 17: Over-voltage Protection Level 3 Block Diagram

This circuit is used to protect the target device from over-voltage and over-current. The over-voltage protection is designed using two zener diodes that face towards each other, across the input and ground. The zener diodes are rated for a breakdown voltage of around 15 Volts, and will break down when the voltage is greater than that. This will short the output to the target device in order to protect against irreparable damage.

The over-current protection circuit is designed to limit the max current to 50mA. The design uses a combination of op-amps and resistors, a shunt resistor, a voltage divider and NMOS transistors to limit the current. The op-amp configuration allows the device to have ±15V. By placing two NMOS transistors back-to-back with the two sources connected together, the current can be limited in both directions. Setting the voltage divider resistance values appropriately allows control over the NMOS transistor to limit the current to 50mA. During simulations, the over-current protection circuit limits the current to 50mA. The simulation schematic is shown in Figures 18 and 19 for -15V and +15V operation. (NU)

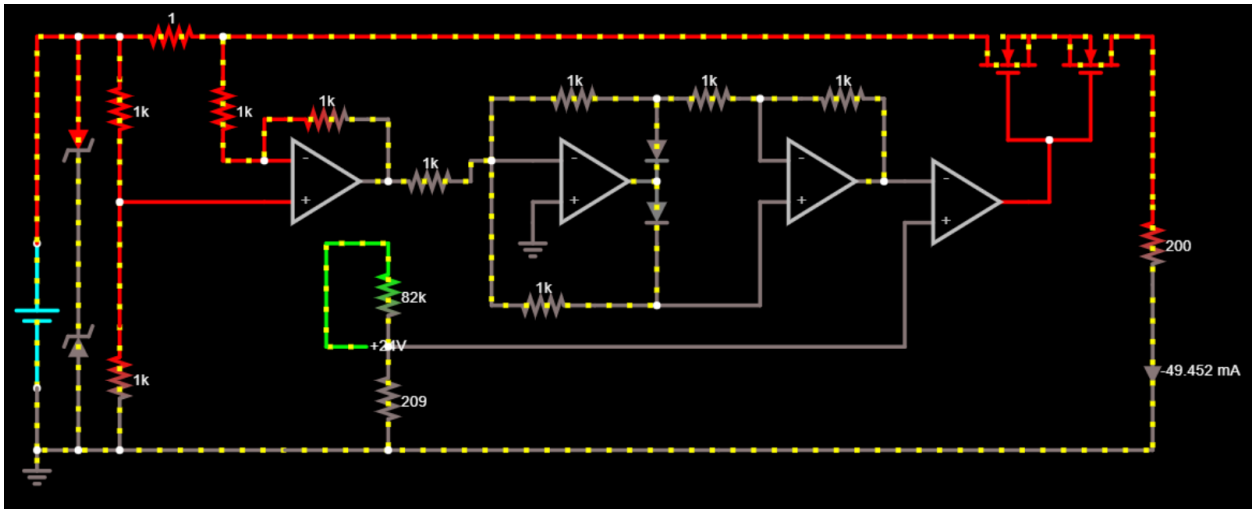


Figure 18: Over-current Protection Simulation, -15V Operation

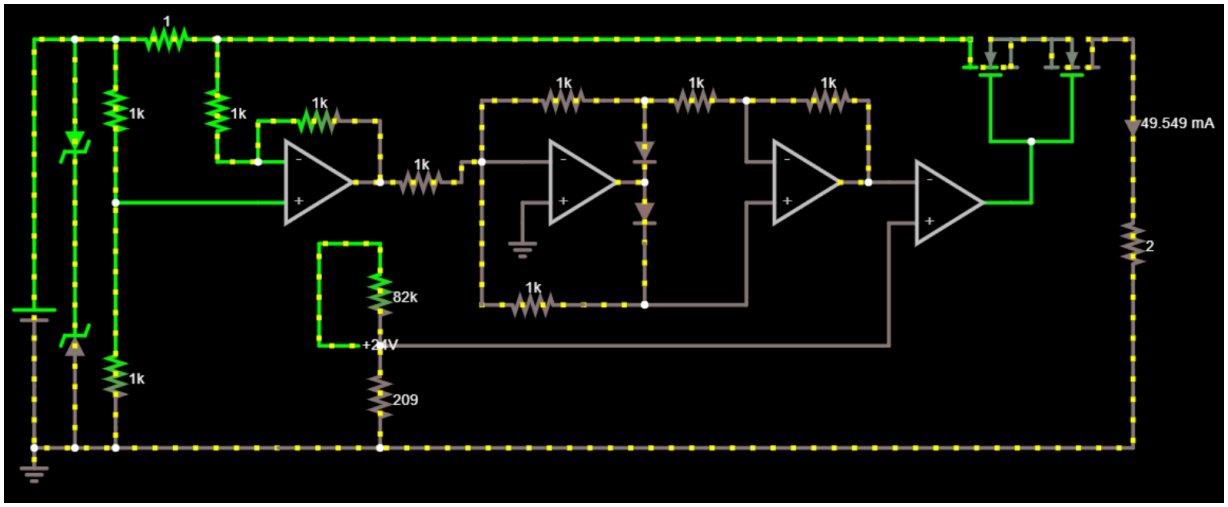


Figure 19: Over-current Protection Simulation, +15V Operation

Tables 30 and 31 list the functional requirements of the blocks shown in Figures 16 and 17.

Table 30: Over-current Protection Functional Requirements

Module	Overcurrent Protection
Designer	Nik Untch
Inputs	- Raw high-voltage digital signals from level shifter - Raw high-voltage digital signals to target device
Outputs	- Current limited to predetermined threshold to board - Current limited to predetermined threshold to target device
Description	The overcurrent protection of the circuit is designed to limit the current going to the I/O Master and the target device. This will help prevent damage due to the board and target device

Table 31: Over-voltage Protection Functional Requirements

Module	Overvoltage Protection
Designer	Nik Untch
Inputs	- Raw high-voltage digital signals from level shifter - Raw high-voltage digital signals to target device
Outputs	- Current limited to predetermined threshold voltage to board - Current limited to predetermined threshold voltage to target device
Description	The overvoltage protection will prevent the user from damaging the board by connecting an incorrect voltage source and reverse polarity

5.2 Hardware Implementation

Previous implementation involved the design of PCBs for each subsystem to demonstrate the validity of each circuit design and act as a platform for further hardware development and improvements. Using the insight gained from these demo boards, a single board was then designed which included all subsystems. This hardware, known as I/O Master v1.0, would then be used to test interoperability between subsystems and further refine the design.

The I/O Master v1.0 hardware consists of 4 I/O pins, a USB 2.0 High Speed interface, STM32H7, and the subsystems required for each I/O pin as outlined above. The PCB is shown in Figure 20. I/O Master v1.0 is OSHW compliant and is released under the MIT license.

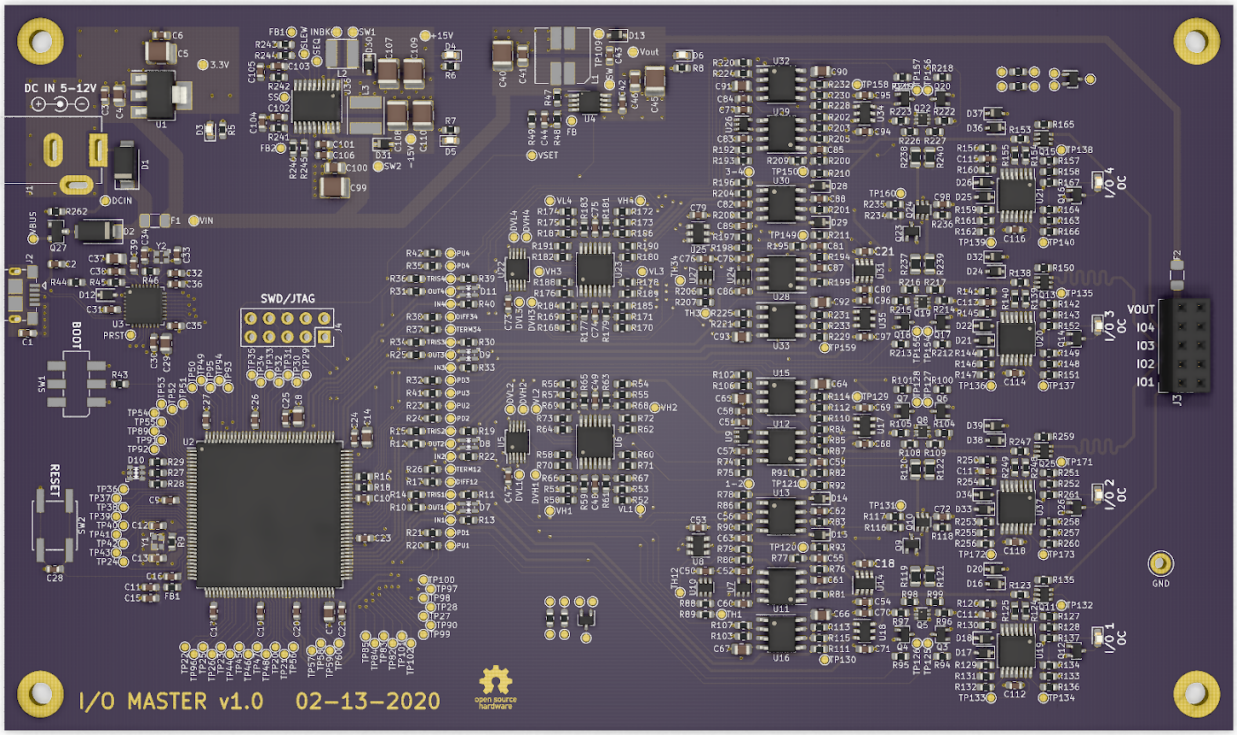


Figure 20: I/O Master v1.0 PCB

The physical hardware consists of a 4-layer PCB, 100 x 170mm, with 468 components. The v1.0 hardware is designed to meet all of the engineering requirements listed in Table 3, not accounting for design bugs or manufacturing errors. Full schematics, PCB layout drawings, and a bill-of-materials are available in the appendix.

During the initial design of the v1.0 hardware, there was a lot of trouble selecting parts that could meet the original ±24V output voltage engineering requirement set earlier. The issue stems from the fact that parts with a higher voltage rating often came with the trade-off of slower performance. This made designing a system that could operate at high speed or high voltage difficult. With the greater availability of parts that could handle -15V to +15V with higher speed and lower cost, the decision was made to lower the requirement to ±15V signals.

5.2.1 Power Inputs

The I/O Master v1.0 can be powered through either the USB port (5V, 500mA) or, optionally, through the DC jack (12V, 500mA) when more power is required. A power switch circuit is used to switch between DC jack power and USB power when they are connected. The power input circuit is shown in Figure 21.

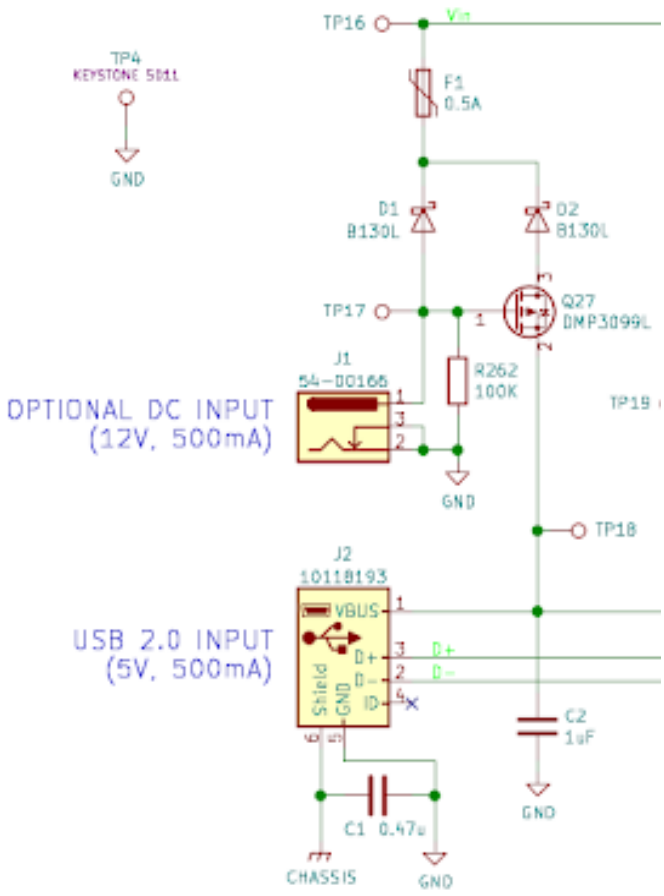


Figure 21: Power Input Circuit

A P-channel FET (Q27) is used to turn off the USB power input when the DC jack is connected, preventing high currents from potentially being drawn from the USB port. Power supply ORing diodes (D1 and D2) are used to prevent either supply from backfeeding if their input voltage differs. A 0.5A PTC resettable fuse (F1) protects the power inputs in the event of a circuit failure on the I/O Master. This type of the fuse allows the I/O Master to continue working after the fault has been cleared, as it resets on its own, and isn't physically damaged when tripped, unlike traditional fuses. (IG)

5.2.2 3.3V Regulator

The 3.3V rail on the board is generated using an AZ1117 linear voltage regulator with a fixed output of 3.3V, shown in Figure 22. Since the 3.3V rail only powers the microcontroller, USB PHY, and configuration logic for the level shifter subsystems, the power requirements for the rail are low. Therefore, a linear regulator was selected to reduce complexity and part count. The specific part chosen is rated for an input voltage range of 4.6-15V with a maximum output current of 800mA.

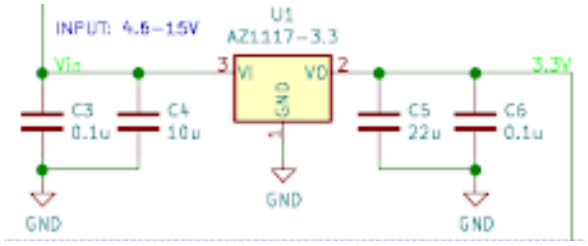


Figure 22: 3.3V Regulator Circuit

Since the power requirements are low, no power calculations were done at the time of schematic design. The PCB layout included large copper areas and thermal vias around the linear regulator to keep the chip cool in the event that there was actually any significant power dissipation. These measures can be seen in Figure 23.

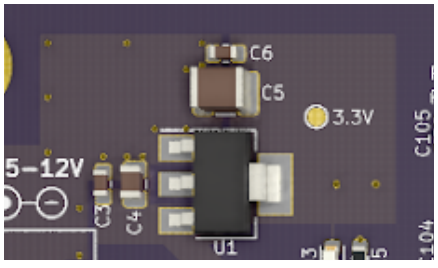


Figure 23: 3.3V Regulator PCB Layout

Once the I/O Master v1.0 hardware could be tested on the bench, the actual power draw of the 3.3V rail could be determined. Some analysis was done to determine the amount of heat dissipated by the linear regulator. Table 32 lists the power draws of the 3.3V rail and calculated power dissipation of the linear regulator under the worst-case scenario (highest input voltage, i.e. when using the 12V DC jack input). Power dissipation of the linear regulator is calculated using the difference in voltage across the regulator (V_{OUT} - V_{IN}) and the current output of the regulator.

Table 32: 3.3V Rail Power Consumption

Microcontroller	0.429 W
RGB LED	0.009 W
USB PHY	0.132 W
Level Shifter DACs	0.009 W
Level Shifter Analog	0.006 W
Level Shifter Biasing	0.001 W
3.3V Rail LED	0.004 W
Total 3.3V Rail Power Draw	**0.590 W**
3.3V Regulator Power Dissipation	**0.304 W**

From the power consumption values, the total current draw on the 3.3V rail is 179mA, which is well within the maximum output current of the regulator. The temperature of the die in the regulator can also be determined using the power dissipation value. The datasheet suggests a thermal resistance of 100°C/W when using a copper area similar to Figure 23. With an ambient temperature of 40°C and a power dissipation of 0.304W, this results in a die temperature of 70.4°C. This is well within the maximum die temperature of 150°C listed in the datasheet. Using

these results, this linear regulator design is within spec and seems to work well for this application. (IG)

5.2.3 ±15V Regulator

Since the power requirements of the +15V and -15V rails are significantly higher than the 3.3V rail, a switching regulator must be used to supply the rails. A switching regulator has significantly greater efficiency and therefore less power dissipation than a linear regulator, at the cost of greater complexity and the introduction of switching noise into the system.

The design used on the ±24V regulator demo board, which utilized the ADP5070, was adapted for ±15V output. The ADP5070 uses a boost topology to generate the positive rail and an inverting topology to generate the negative rail. The existing regulator design was modified to use smaller components, as the current requirements on each rail were determined to be less than what was designed for on the demo board.

The overall regulator schematic/internal block diagram from the datasheet is shown in Figure 24.

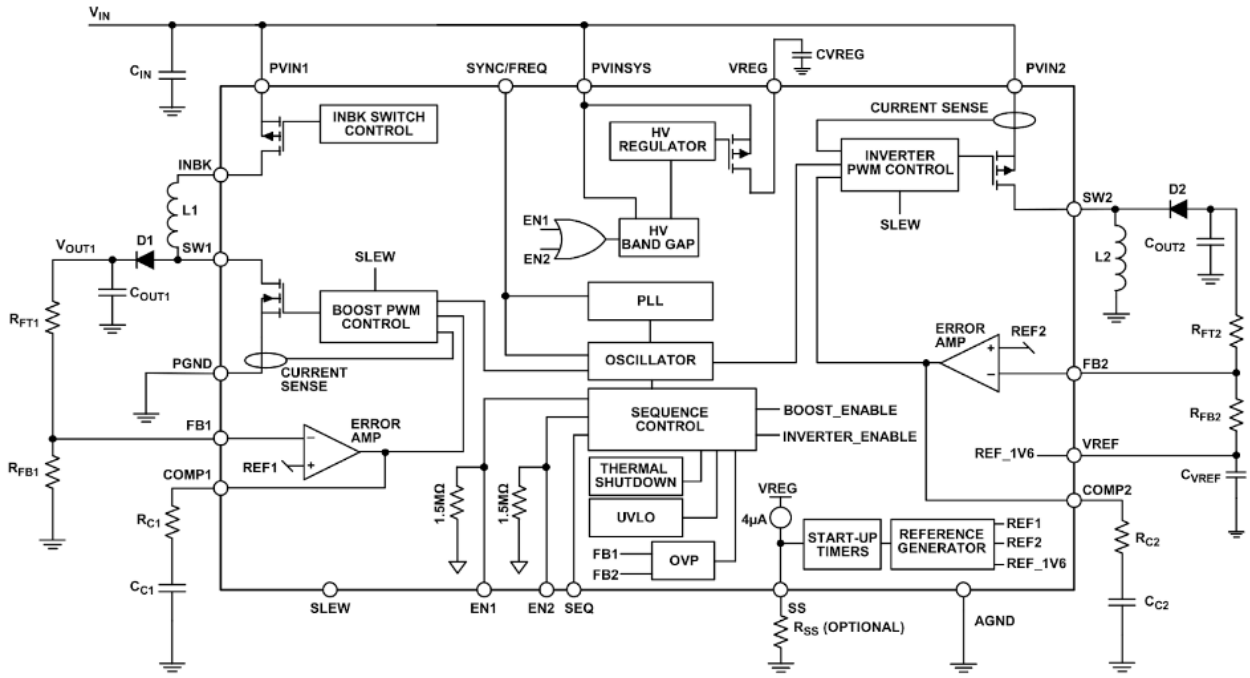


Figure 24: ADP5070 Datasheet Regulator Schematic / Internal Block Diagram

For each rail, the regulator was designed for an average inductor ripple current of approximately 30% of the total output current. This is a common rule-of-thumb in switching regulator design, as it is a good balance of minimizing component size, efficiency, and output voltage ripple. The switching frequency of 1.2MHz was selected in order to minimize component size at the cost of some additional power dissipation. The diodes D1 and D2 were selected for low forward voltage drop, low junction capacitance, and a high power dissipation rating in order to account for the output current rating. The inductors L1 and L2 were selected to minimize size, shielded construction to minimize stray field, and high enough current ratings in order to not saturate at the required output current. In addition, the datasheet recommended adding loop compensation components in order to cancel out non-idealities in the regulator IC and feedback loop. These

non-idealities result in right-half-plane zeros in the system which contribute to instability. Design values for each voltage rail at each input voltage condition are listed in Tables 33 and 34.

Table 33: +15V Output Switching Regulator Design Values

		V_{IN} = 5V	V_{IN} = 12V	
Design Parameters				
Output Voltage	V_{OUT}		15	V
Comparator Reference	V_{REF}		0.8	V
Output Current	I_{OUT}		0.400	A
Switching Frequency	f_{SW}		1.2	MHz
D1 Part No.			NSR0530HT1G	
D1 Forward Voltage	V_F		0.62	V
D1 Junction Capacitance	C_J		10	pF
L1 Part No.			LQH3NPN100MMEL	
L1 Inductance	L		10	µH
Calculated Inductor Values				
Duty Cycle	D	67.99	23.18	%
Input Current	I_{IN}	1.250	0.521	A
On Time	t_{ON}	0.567	0.193	µs
Ripple Current	I_{L1}	0.283	0.232	A
Ripple Current (%)	I_{L1}	22.67	44.51	%
Peak Current	I_{L1}	1.391	0.637	A
L1 Minimum Inductance	$L_{1\text{-}MIN}$	2.567	0.257	µH
Calculated Loop Compensation Values				
Zero Frequency	$f_{Z\text{-}RHP}$	61.155	207.000	kHz
Compensation Resistance	R_C	115.25	162.55	KΩ
Compensation Capacitance	C_C	903	189	pF

Table 34: -15V Output Switching Regulator Design Values

		V_{IN} = 5V	V_{IN} = 12V	
Design Parameters				
Output Voltage	V_{OUT}		-15	V
Comparator Reference	V_{REF}		0.8	V
Output Current	I_{OUT}		0.400	A
Switching Frequency	f_{SW}		1.2	MHz
D2 Part No.			NSR0530HT1G	
D2 Forward Voltage	V_F		0.62	V
D2 Junction Capacitance	C_J		10	pF
L2 Part No.			ASPI-4030S-150M-T	
L2 Inductance	L		15	µH
Calculated Inductor Values				
Duty Cycle	D	75.75	56.55	%
Input Current	I_{IN}	1.650	0.921	A
On Time	t_{ON}	0.631	0.471	µs
Ripple Current	I_{L2}	0.210	0.377	A
Ripple Current (%)	I_{L2}	12.76	40.95	%
Peak Current	I_{L2}	1.755	1.109	A
L2 Minimum Inductance	$L_{2\text{-}MIN}$	3.917	3.497	µH
Calculated Loop Compensation Values				
Zero Frequency	$f_{Z\text{-}RHP}$	8.000	8.000	kHz
Compensation Resistance	R_C	3.52	1.76	KΩ
Compensation Capacitance	C_C	0.226	0.452	µF

The final schematic of the switching regulator is shown in Figure 25.

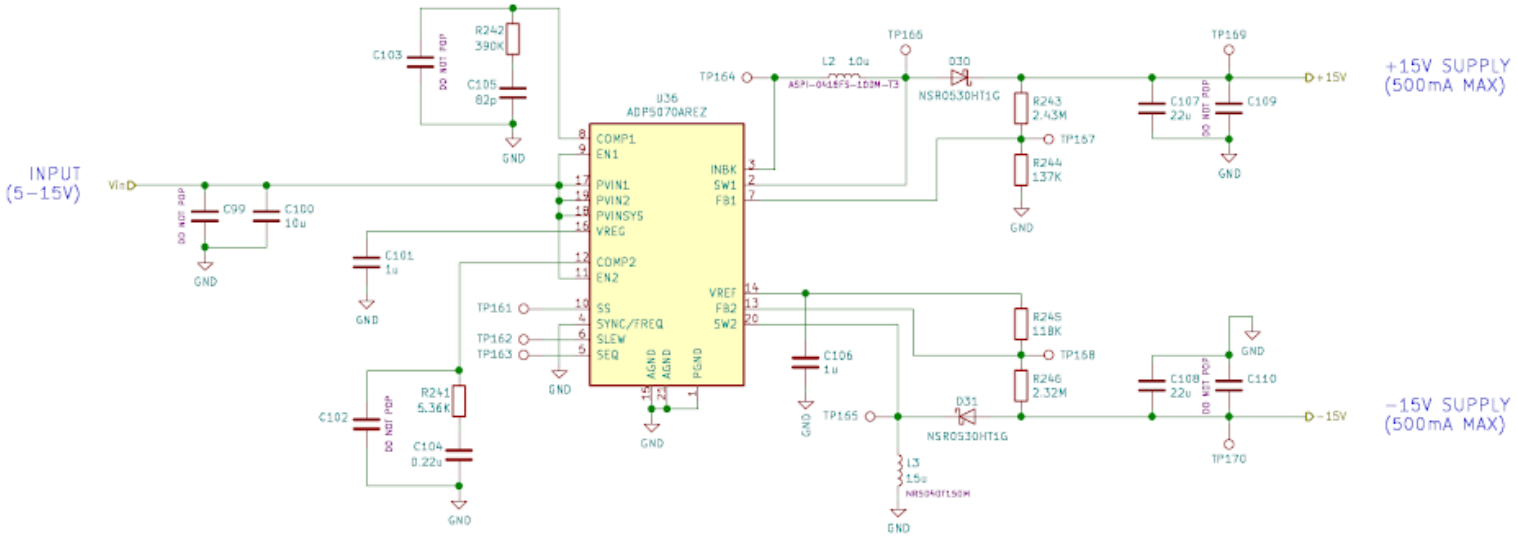


Figure 25: Final ±15V Regulator Schematic

The PCB layout of the switching regulator had to be carefully considered in order to ensure that the regulator would operate properly. The feedback loop formed between the regulator IC, inductor, diode, and feedback resistors must be as small as possible in order to reduce the amount of parasitic inductance that is introduced into the loop. Parasitic inductance translates into delay in the feedback loop, which can reduce the performance of the regulator and cause instability. Figure 26 shows the recommended layout from the datasheet, and Figure 27 shows the actual layout of the ±15V regulator circuit. (IG)

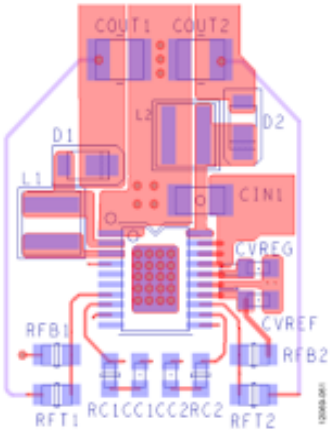


Figure 26: AD5070 Datasheet Recommended Layout

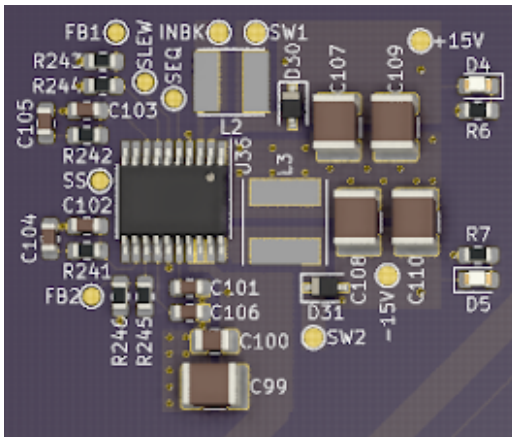


Figure 27: ±15V Regulator PCB Layout

5.2.4 3.3-15V Adjustable Regulator

The adjustable regulator was designed to be able to supply power to the target device. Since each device connected to the I/O Master could potentially operate at a different voltage, having an adjustable regulator was essential. Similar to the ±15V Regulator, the power output requirements resulted in the use of a switching regulator.

The LM2621 was selected for use in a SEPIC topology, which can step-up or step-down the input voltage. The SEPIC topology allows a single, dual winding, inductor to be used in place of two separate inductors, reducing the required component area on the PCB. The rated input voltage range is 1.4 to 14V, which is acceptable given that the actual input voltage is 5-12V. The schematic provided in the datasheet for the LM2621 is represented in Figure 28.

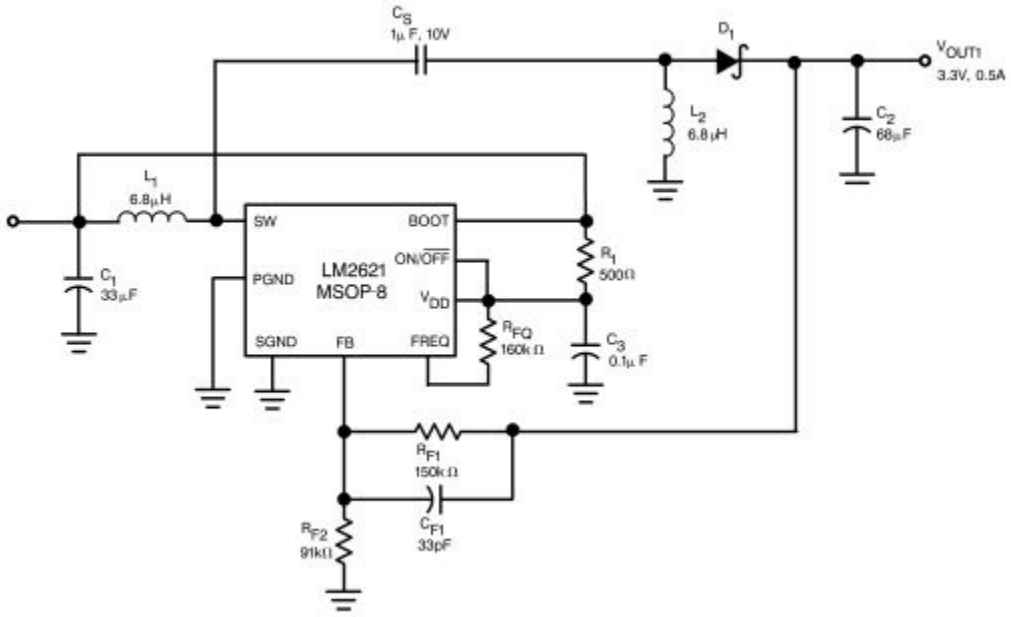


Figure 28: LM2621 Datasheet SEPIC Regulator Schematic

The design values for the regulator are listed in Tables 35 and 36.

Table 35: Adjustable Switching Regulator Design Values, V_{OUT} = 3.3V

		V_{IN} = 5V	V_{IN} = 12V	
Design Parameters				
Output Voltage	V_{OUT}		3.3	V
Comparator Reference	V_{REF}		0.8	V
Output Current	I_{OUT}		0.100	A
Switching Frequency	f_{SW}		1.2	MHz
Bootstrap Capacitance	C_S		1	µF
Output Capacitance	C_{OUT}		10	µF
D1 Part No.			NSR0530HT1G	
D1 Forward Voltage	V_F		0.62	V
D1 Junction Capacitance	C_J		10	pF
L1-L2 Part No.			SRF0703-820M	
L1-L2 Winding Inductance	L		82	µH
Calculated Inductor Values				
Duty Cycle	D	43.95	24.62	%
Input Current	I_{IN}	0.078	0.033	A
On Time	t_{ON}	0.366	0.205	µs
Ripple Current	I_{L1}	0.011	0.015	A
Ripple Current (%)	I_{L1}	14.24	45.96	%
Peak Current	I_{L1}	0.084	0.040	A
Calculated Diode Values				
Dissipated Power	P_D	0.062	0.062	W
Peak Current	I_D	0.168	0.080	A
Calculated Capacitor Values				
Input Capacitor Ripple Current	$I_{IN\text{-}RMS}$	0.003	0.004	A
Output Capacitor Ripple Current	$I_{OUT\text{-}RMS}$	0.089	0.057	A
Calculated Loop Compensation Values				
Zero Frequency	$f_{Z\text{-}RHP}$	20.125	36.391	kHz
Compensation Resistance	R_C	18.88	1.38	KΩ
Compensation Capacitance	C_C	0.172	0.126	µF

Table 36: Adjustable Switching Regulator Design Values, V_{OUT} = 15V

		V_{IN} = 5V	V_{IN} = 12V	
Design Parameters				
Output Voltage	V_{OUT}		15	V
Comparator Reference	V_{REF}		0.8	V
Output Current	I_{OUT}		0.100	A
Switching Frequency	f_{SW}		1.2	MHz
Bootstrap Capacitance	C_S		1	µF
Output Capacitance	C_{OUT}		10	µF
D1 Part No.			NSR0530HT1G	
D1 Forward Voltage	V_F		0.62	V
D1 Junction Capacitance	C_J		10	pF
L1-L2 Part No.			SRF0703-820M	
L1-L2 Winding Inductance	L		82	µH
Calculated Inductor Values				
Duty Cycle	D	75.75	56.55	%
Input Current	I_{IN}	0.312	0.130	A
On Time	t_{ON}	0.631	0.471	µs
Ripple Current	I_{L1}	0.019	0.034	A
Ripple Current (%)	I_{L1}	6.16	26.49	%
Peak Current	I_{L1}	0.322	0.147	A
Calculated Diode Values				
Dissipated Power	P_D	0.062	0.062	W
Peak Current	I_D	0.644	0.295	A
Calculated Capacitor Values				
Input Capacitor Ripple Current	$I_{IN\text{-}RMS}$	0.006	0.010	A
Output Capacitor Ripple Current	$I_{OUT\text{-}RMS}$	0.177	0.114	A
Calculated Loop Compensation Values				
Zero Frequency	$f_{Z\text{-}RHP}$	17.118	54.956	kHz
Compensation Resistance	R_C	32.26	43.15	KΩ
Compensation Capacitance	C_C	0.011	0.002	µF

The adjustable regulator receives a control signal from the microcontroller in order to set the output voltage. Via the GUI, the user selects the desired voltage for powering the target device. To adjust the output voltage, the microcontroller applies a 0-3.3V signal to the bottom of the feedback voltage divider using the built-in DAC. This shifts the feedback input to the regulator control loop, which proportionally changes the output of the regulator accordingly. The regulator output voltage vs. microcontroller DAC voltage relationship can be seen in Figure 29.

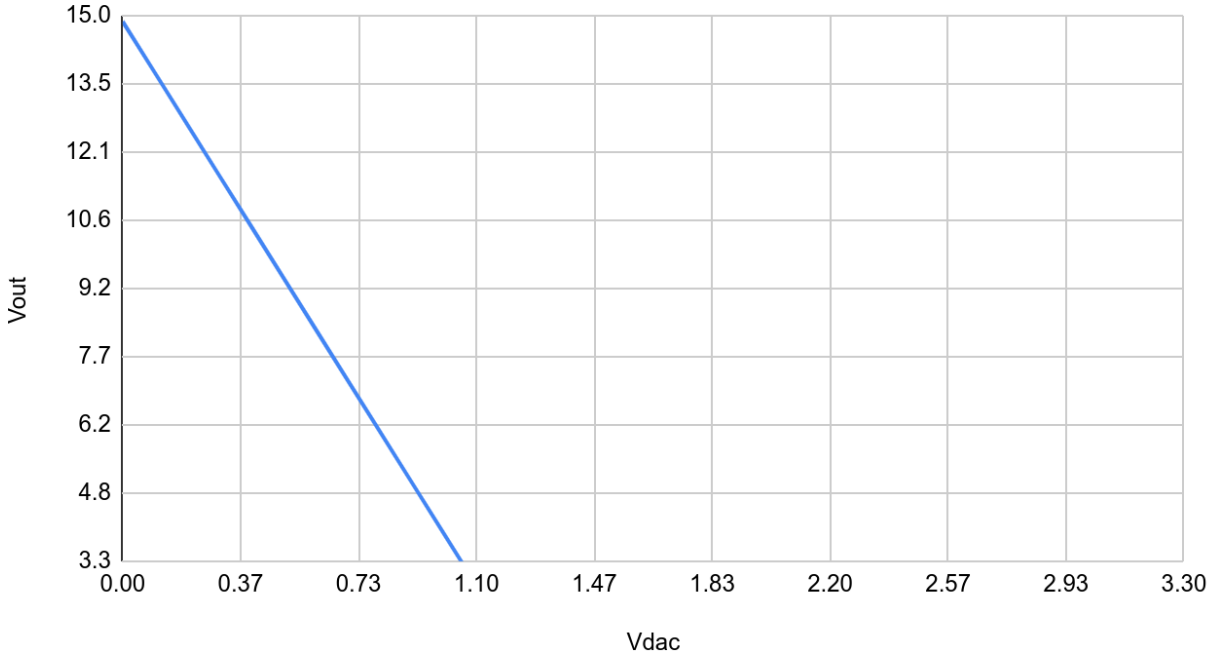


Figure 29: Adjustable Regulator Output Voltage vs. Microcontroller DAC Voltage

The final schematic of the regulator with DAC-controlled output voltage is shown in Figure 30. A 0.1A PTC fuse was added to the adjustable regulator output to protect against short circuits on the target device power output. (IG)

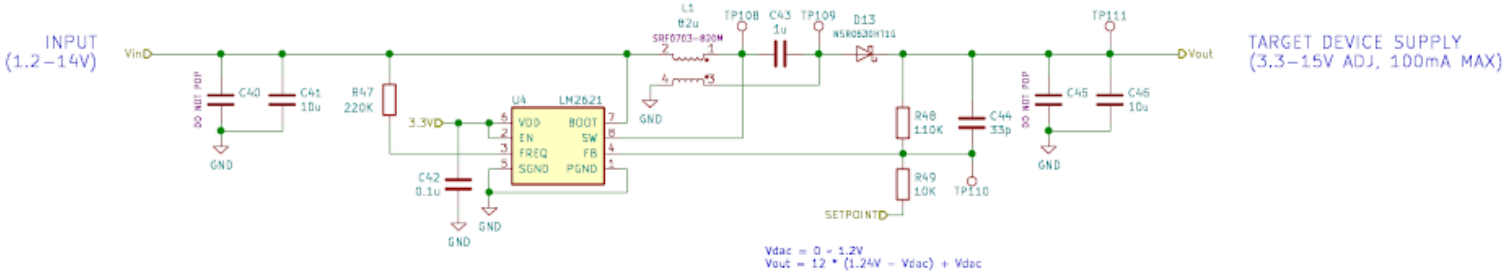


Figure 30: Final 3.3-15V Adjustable Regulator Schematic

Like the ±15V Regulator, the PCB layout of the regulator is critical to its operation. The recommended layout from the LM2621 datasheet is shown in Figure 31.

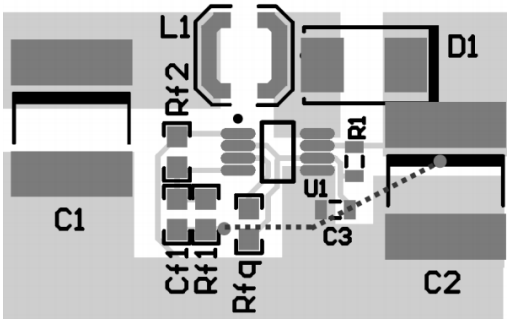


Figure 31: LM2621 Recommended PCB Layout

Figure 32 shows the actual layout of the adjustable regulator on the I/O Master v1.0 hardware.

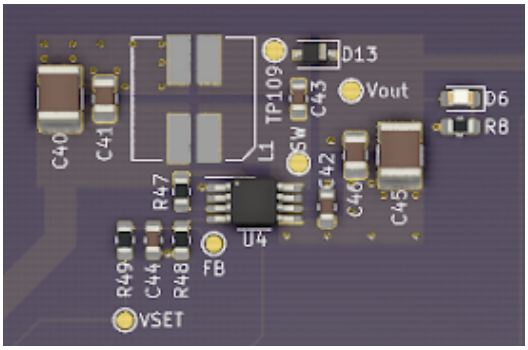


Figure 32: Adjustable Regulator PCB Layout

5.2.5 Microcontroller

The STM32H7 microcontroller has specific requirements for supplying power and decoupling capacitors. Figure 33 shows these requirements as listed in the datasheet.

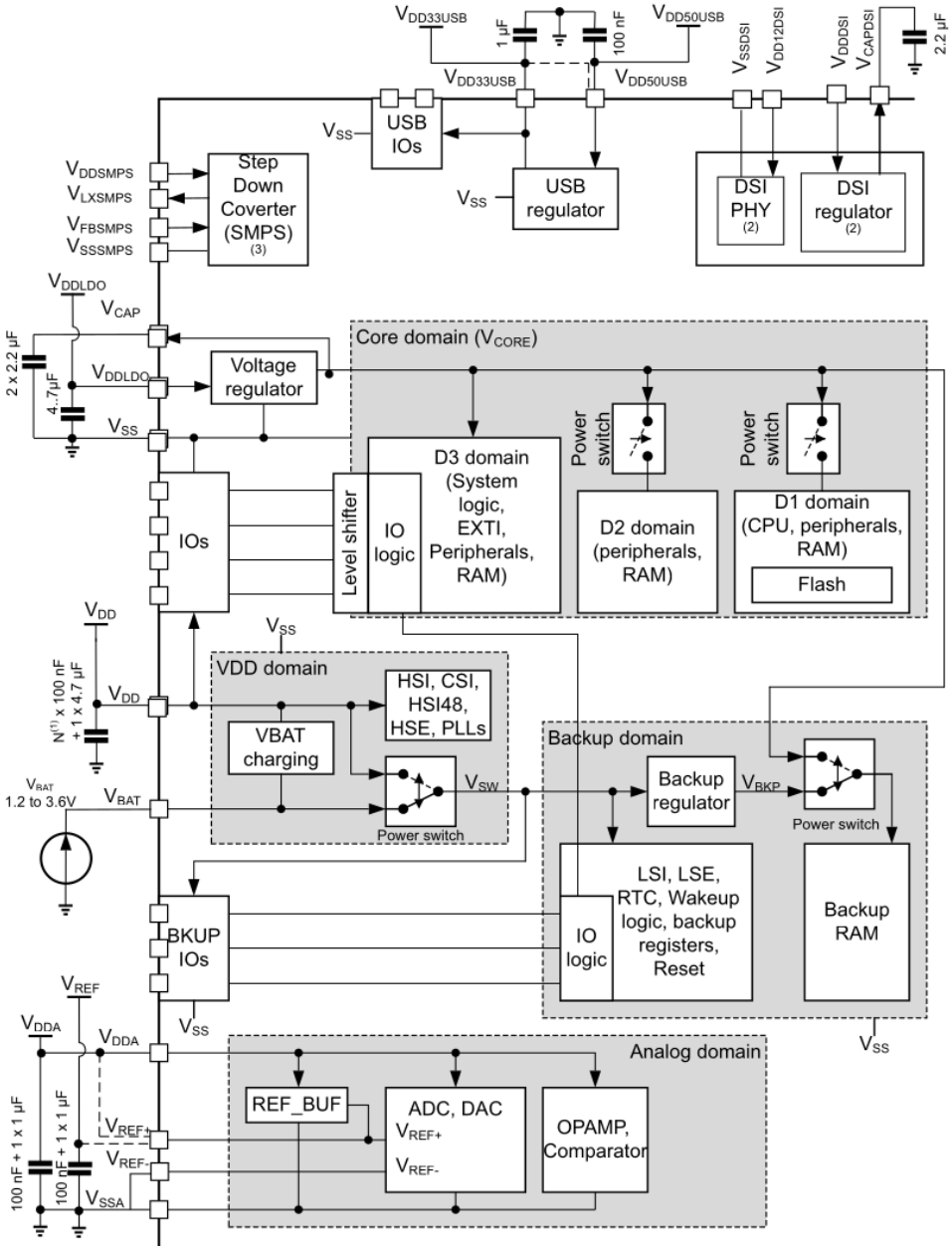


Figure 33: STM32H7 Datasheet Power and Decoupling Requirements

The VDD, VDD3_USB, and VDDA pins are supplied from the 3.3V regulator. Each VDD pin gets a 0.1uF decoupling capacitor along with one 4.7uF capacitor for the group of VDD pins. The VDD3_USB pin gets a 1uF capacitor. The VDDA pin is supplied with 3.3V through a ferrite bead in order to prevent noise from reaching the microcontroller's ADCs, and it gets a 0.1uF capacitor along with a 1uF capacitor. The VCAP1 and VCAP2 pins are not supplied any voltage and each get a 2.2uF capacitor in order to decouple the internal voltage regulator.

An external 48MHz crystal oscillator is used to provide the microcontroller with a high speed clock source that is more accurate and stable than the internal clock source. High speed peripherals such as the USB interface require the accurate clock signal. The crystal is loaded with capacitors that have been calculated to match the internal load capacitance of the oscillator on the chip to ensure stable operation. Figure 34 shows the schematic of the external oscillator.

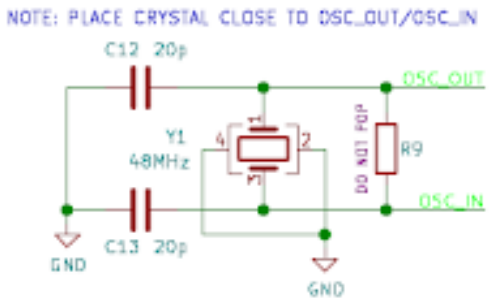


Figure 34: Microcontroller External Crystal Schematic

The specific microcontroller that was selected was the STM32H743ZI in a 144-pin LQFP package. The function of each pin must be carefully decided in a way that all required microcontroller peripherals can be utilized in a way without overlapping pin assignments. Peripherals can be assigned to different pins in software, however a specific peripheral can only

be assigned to a certain subset of pins, depending on the peripheral. For I/O Master, many peripherals are used and nearly all pins on the 144-pin package are utilized. The peripherals are:

- Three GPIO ports (8 pins each) must be reserved for outputs, tristate mode, and inputs to/from the level shifters.
- A hardware clock and corresponding output pin must be reserved for each I/O pin. These are used to generate clock signals without using CPU resources.
- Two GPIO pins must be reserved for each I/O pin for configuring pull-up and pull-down resistors.
- Two GPIO pins must be reserved for each pair of I/O pins for configuring differential mode and termination resistors.
- Three GPIO pins with PWM capability must be reserved for the RGB status LED.
- An I2C interface must be reserved for communication to the level shifter DACs.
- A UART interface must be reserved for debugging.
- The built-in USB interface must be reserved for debugging.
- The ULPI interface pins must be reserved for the USB PHY
- The Serial Wire Debug interface pins must be reserved for programming/debugging the microcontroller.

The final pin assignments and their corresponding peripherals are listed in Table 37. (IG)

Table 37: STM32H7 Microcontroller Pin Assignments

Pin #	Name	Peripheral	Signal
1	PE2		*Reserved, Do Not Connect*
2	PE3		*Reserved, Do Not Connect*
3	PE4		*Reserved, Do Not Connect*
4	PE5		*Reserved, Do Not Connect*
5	PE6		*Reserved, Do Not Connect*
7	PC13	GPIO Output	*IO_7&8_DIFF (reserved for v2)*
8	PC14/OSC32_IN	GPIO Output	*IO_5&6_DIFF (reserved for v2)*
9	PC15/OSC32_OUT	GPIO Output	*~IO_5&6_TERM (reserved for v2)*
10	PF0	GPIO Output	STATUS_B
11	PF1	GPIO Output	STATUS_R
12	PF2	GPIO Output	STATUS_G
13	PF3		*Unused*
14	PF4		*Unused*
15	PF5		*Unused*
18	PF6		*Unused*
19	PF7		*Unused*
20	PF8		*Unused*
21	PF9		*Unused*
22	PF10		*Unused*
26	PC0	ULPI	ULPI_STP
27	PC1	GPIO Output	*IO_7_PU (reserved for v2)*
28	PC2_C	ULPI	ULPI_DIR
29	PC3_C	ULPI	ULPI_NXT
34	PA0	Timer 2	IO_1_CLK
35	PA1	Timer 5	IO_2_CLK
36	PA2	Timer 15	IO_3_CLK
37	PA3	ULPI	ULPI_D0
40	PA4	DAC	Vout_SETPOINT
41	PA5	ULPI	ULPI_CK
42	PA6	GPIO Output	*IO_6_PU (reserved for v2)*
43	PA7	Timer 14	*IO_8_OUT (reserved for v2)*

44	PC4	GPIO Output	*~IO_7_PD (reserved for v2)*
45	PC5	GPIO Output	*IO_8_PU (reserved for v2)*
46	PB0	ULPI	ULPI_D1
47	PB1	ULPI	ULPI_D2
48	PB2	GPIO Output	*~IO_6_PD (reserved for v2)*
49	PF11		*Unused*
50	PF12		*Unused*
53	PF13		*Unused*
54	PF14		*Unused*
55	PF15		*Unused*
56	PG0	GPIO Output	*IO_5_PU (reserved for v2)*
57	PG1	GPIO Output	*~IO_5_PD (reserved for v2)*
58	PE7		*Reserved, Do Not Connect*
59	PE8	GPIO Input	IO_1_IN
60	PE9	GPIO Input	IO_2_IN
63	PE10	GPIO Input	IO_3_IN
64	PE11	GPIO Input	IO_4_IN
65	PE12	GPIO Input	*IO_5_IN (reserved for v2)*
66	PE13	GPIO Input	*IO_6_IN (reserved for v2)*
67	PE14	GPIO Input	*IO_7_IN (reserved for v2)*
68	PE15	GPIO Input	*IO_8_IN (reserved for v2)*
69	PB10	ULPI	ULPI_D3
70	PB11	ULPI	ULPI_D4
73	PB12	ULPI	ULPI_D5
74	PB13	ULPI	ULPI_D6
75	PB14	GPIO Output	IO_1_PU
76	PB15	GPIO Output	~IO_1_PD
77	PD8	GPIO Output	~IO_1_TRIS
78	PD9	GPIO Output	~IO_2_TRIS
79	PD10	GPIO Output	~IO_3_TRIS
80	PD11	GPIO Output	~IO_4_TRIS
81	PD12	GPIO Output	*~IO_5_TRIS (reserved for v2)*
82	PD13	GPIO Output	*~IO_6_TRIS (reserved for v2)*
85	PD14	GPIO Output	*~IO_7_TRIS (reserved for v2)*

86	PD15	GPIO Output	*~IO_8_TRIS (reserved for v2)*
87	PG2	GPIO Output	*IO_1&2_DIFF*
88	PG3	GPIO Output	*~IO_1&2_TERM*
89	PG4	GPIO Output	*~IO_2_PD*
90	PG5	GPIO Output	*IO_2_PU*
91	PG6	GPIO Output	*IO_3_PU*
92	PG7	GPIO Output	*~IO_3_PD*
93	PG8		*Unused*
96	PC6	Timer 3	IO_4_CLK
97	PC7	GPIO Output	*~IO_8_PD (reserved for v2)*
98	PC8	GPIO Output	*~IO_7&8_TERM (reserved for v2)*
99	PC9	I2C 3	I2C_SDA
100	PA8	I2C 3	I2C_SCL
101	PA9	USART 1	UART_TXO
102	PA10	USART 1	UART_RXI
103	PA11	USB	USB_FS_DM
104	PA12	USB	USB_FS_DP
105	PA13	SWD	TMS/SWDIO
109	PA14	SWD	TCK/SWDCLK
110	PA15	SWD	TDI
111	PC10	GPIO Output	~IO_3&4_TERM
112	PC11	GPIO Output	IO_3&4_DIFF
113	PC12	GPIO Output	~IO_4_PD
114	PD0	GPIO Output	IO_1_OUT
115	PD1	GPIO Output	IO_2_OUT
116	PD2	GPIO Output	IO_3_OUT
117	PD3	GPIO Output	IO_4_OUT
118	PD4	GPIO Output	*IO_5_OUT (reserved for v2)*
119	PD5	GPIO Output	*IO_6_OUT (reserved for v2)*
122	PD6	GPIO Output	*IO_7_OUT (reserved for v2)*
123	PD7	GPIO Output	*IO_8_OUT (reserved for v2)*
124	PG9		*Unused*
125	PG10		*Unused*
126	PG11		*Unused*

127	PG12		*Unused*
128	PG13		*Unused*
129	PG14		*Unused*
132	PG15		*Unused*
133	PB3	SWD	TDO/SWO
134	PB4	SWD	nRESET
135	PB5	ULPI	ULPI_D7
136	PB6	GPIO Output	IO_4_PU
137	PB7	Timer 4	*IO_5_OUT (reserved for v2)*
139	PB8	Timer 16	*IO_6_OUT (reserved for v2)*
140	PB9	Timer 17	*IO_7_OUT (reserved for v2)*
141	PE0		*Reserved, Do Not Connect*
142	PE1		*Reserved, Do Not Connect*

Figure 36 shows the schematic of the microcontroller with those pin assignments.

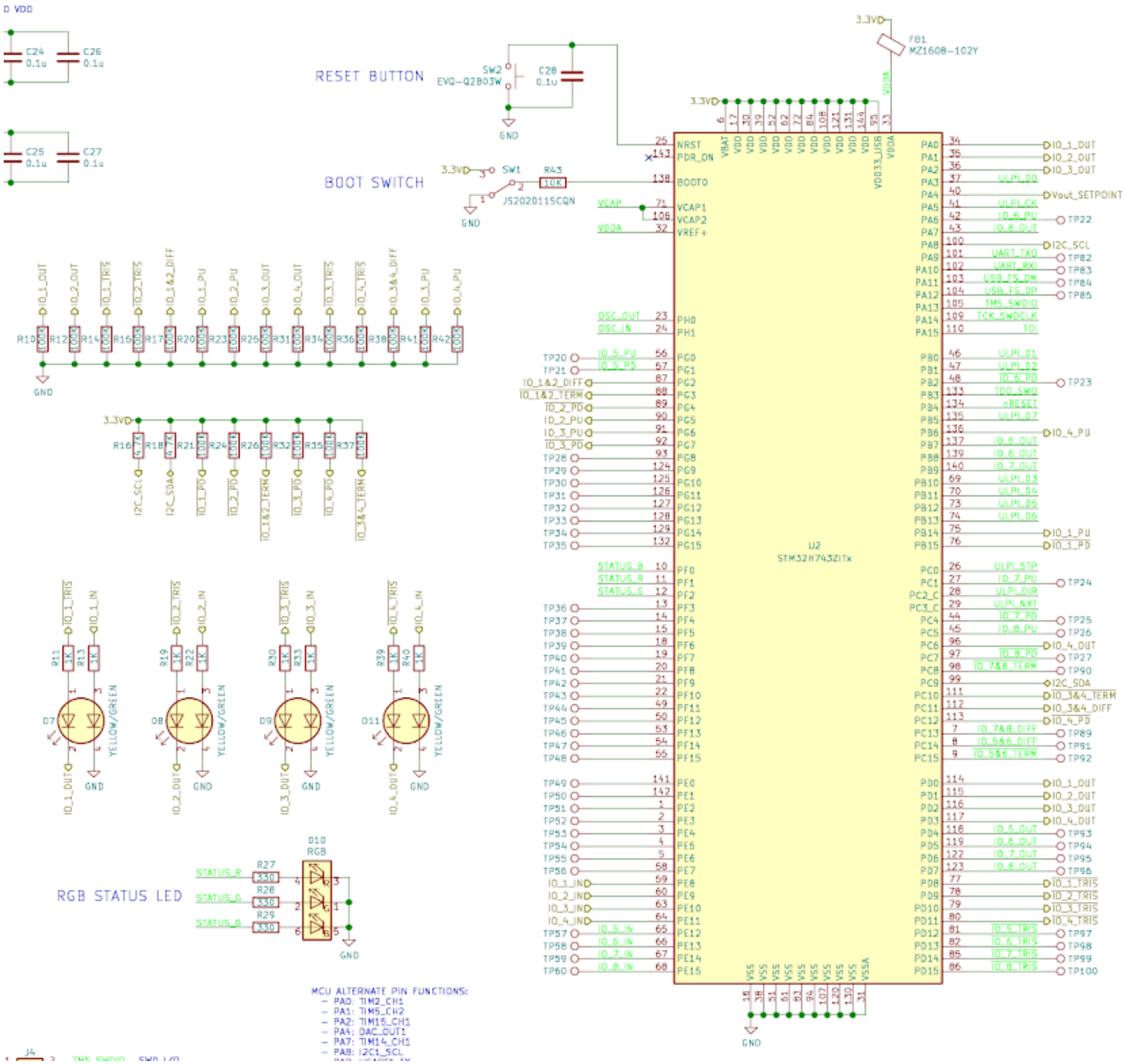


Figure 35: Microcontroller GPIO Schematic

Pull-up and pull-down resistors were added on many GPIO pins in order to force a default logic state when the microcontroller is not driving the pin. Since the I/O Master v1.0 hardware is a

prototype, test points were added on many critical signals so that they could be more easily probed. Dual-color LEDs were added on level shifter communication pins to allow the user to visually see when I/O pins are in use. An orange LED is used on output pins and a green LED is used on input pins. An RGB status LED was added to the microcontroller so that the user can determine the current status of the firmware and ensure that the I/O Master continues to operate. The I/O status LEDs on inputs and outputs can be seen in Figure 36 (the four orange LEDs in the center of the photo). (IG)

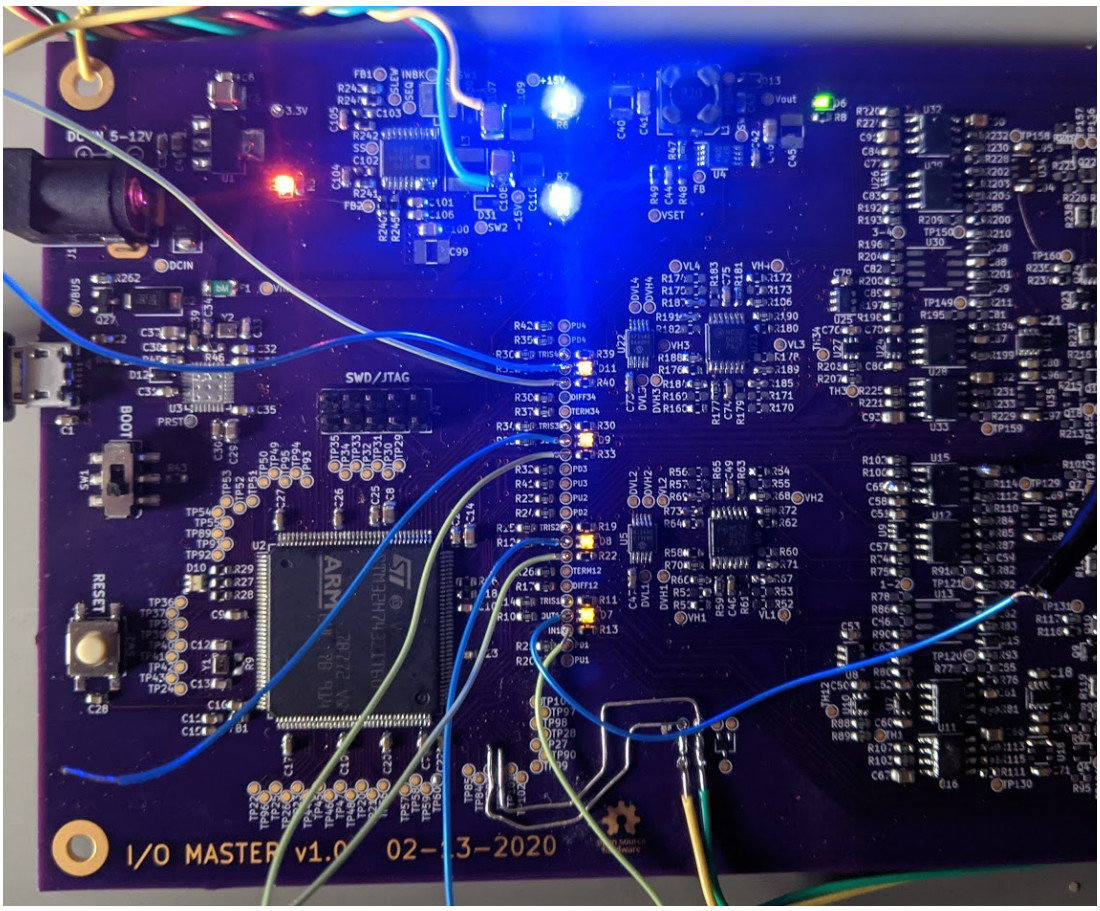


Figure 36: I/O Master Status LEDs
(Top Left: 3.3V Rail Status, Top Center: +15V and -15V Rail Statuses,
Top Right: Adjustable Regulator Status, Center: I/O Pin Activity)

5.2.6 USB Interface

Since the I/O Master can stream data at up to 20MBps (20MHz) to/from the target device, a high speed interface is needed to communicate with the computer. The USB interface built-in to the STM32H7 microcontroller only supports up to USB 1.1 Full Speed, which has a theoretical max speed of 12Mbps. To use USB 2.0 (480Mbps theoretical max speed), the USB3300 external USB transceiver was chosen. This chip is also known as a USB PHY. It communicates with the microcontroller over a high-speed ULPI interface and handles physically driving the USB port. A block diagram from the USB3300 datasheet is shown in Figure 37. (NU)

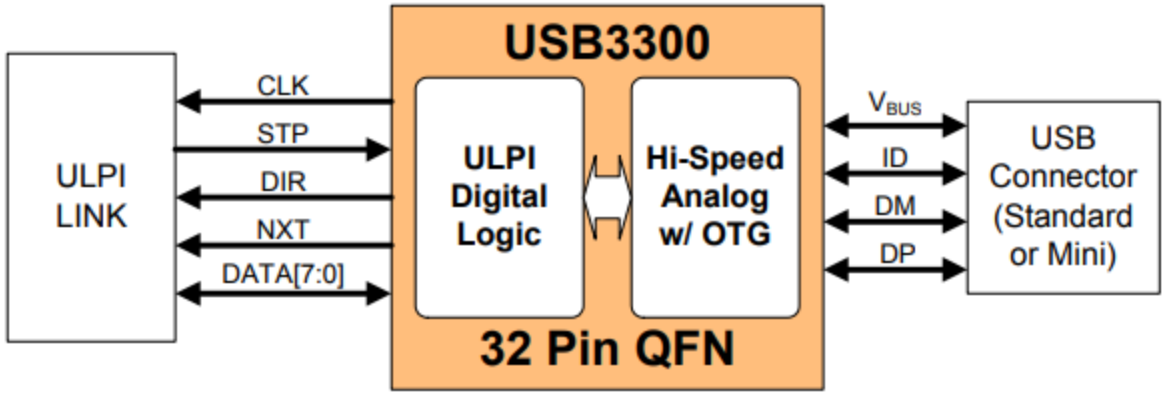


Figure 37: USB3300 Datasheet Block Diagram

The schematic for the USB330 is shown in Figure 38.

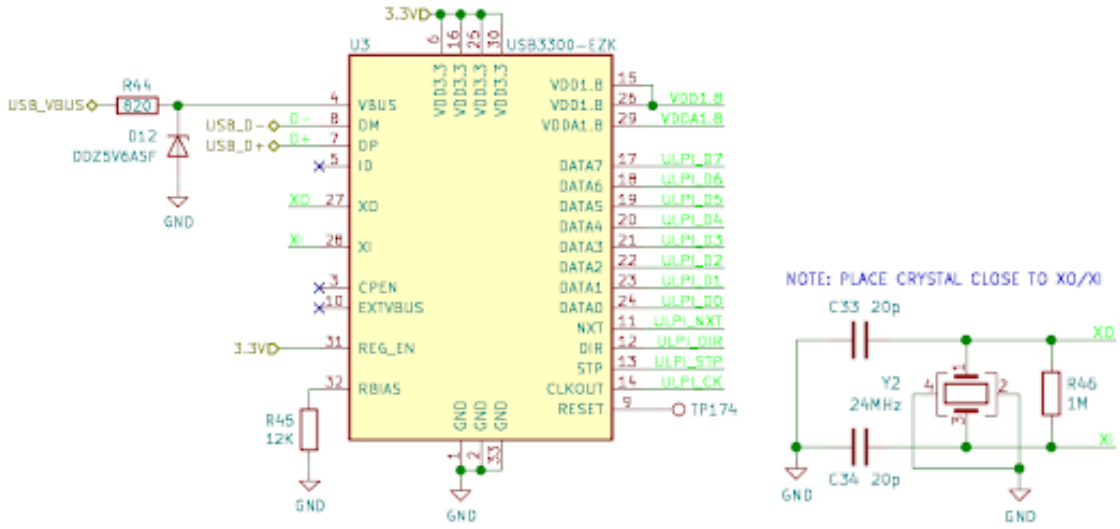


Figure 38: USB3300 USB PHY Schematic

The PCB layout for the USB PHY requires some consideration for the high-speed signalling. The USB D+ and D- signals must be differentially routed to the USB port and the PHY chip must physically be close to the USB port and the microcontroller. Figure 39 shows the actual layout of the USB PHY interface.

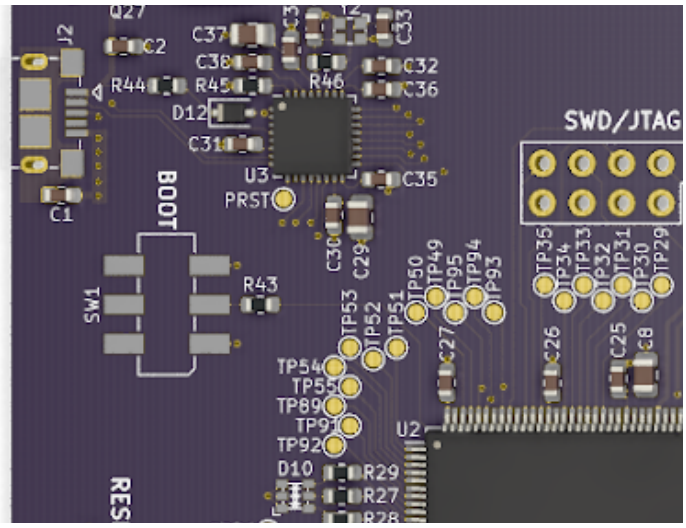


Figure 39: USB PHY PCB Layout

5.2.7 Level Shifter

For logic level generation, the MCP4728 DAC was selected. This is a 4 channel, 12-bit DAC with I2C interface. With the output scaled to ±15V, this provides a resolution of 7.3mV per step. This is more than adequate, as logic levels are typically specified with no more than 100mV resolution. Using the level shifter demo board as a starting point, the op-amp feedback network was redesigned for ±15V rails. This results in the following relationship between DAC output voltage and logic level voltage:

$$V_{LOGIC-LEVEL} = \frac{1000}{110}(1.65 - V_{DAC})$$

Note that the scaling operation is inverted. With ±15V rails, this scaling ratio can be perfectly represented with common nominal resistor values, providing more accuracy than the demo board with ±24V rails. The NCV20074 was selected as an op-amp due to its wide supply range of 2.7 to 36V (single-ended), small 4-op-amp package, and low cost. Figure 40 shows a portion of the logic level generator schematic.

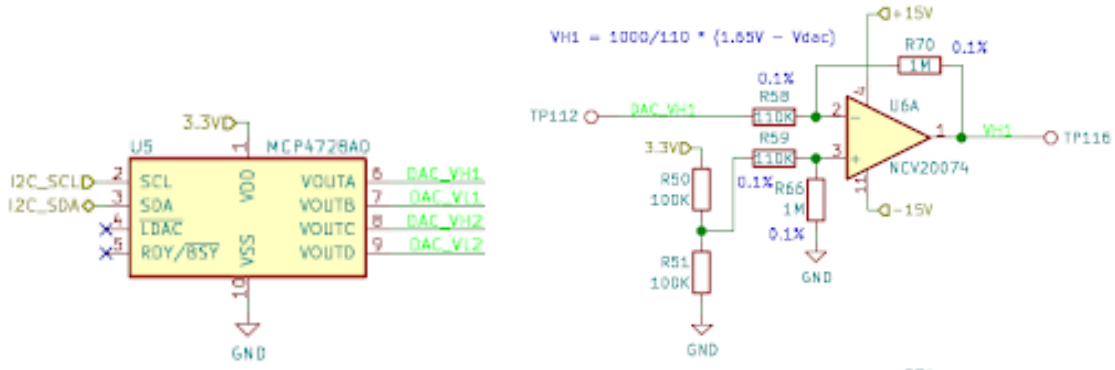


Figure 40: Logic Level Generator Schematic

The original design for the output driver consisted of a push-pull FET stage. Further research and simulations showed that while a high-speed push-pull stage could be implemented, high-speed and high-voltage operation would require a complex FET driver circuit with no guarantee that it would work first try. At speeds up to 10MHz, this setup would be costly, have a large number of components, and be very sensitive to PCB layout.

A simpler op-amp circuit was chosen as an alternative. This circuit is shown in Figure 41.

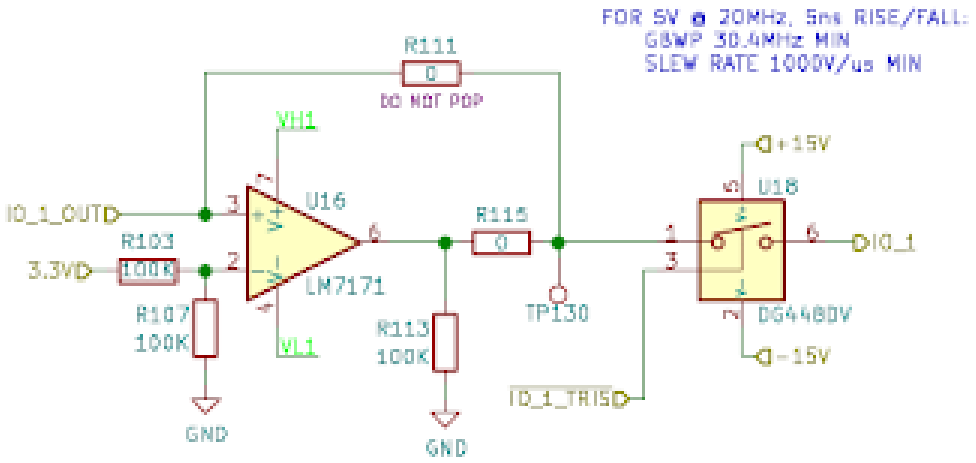


Figure 41: Output Driver Schematic

This specific circuit is similar to a comparator, where the op-amp operates in open-loop mode and the microcontroller signal is compared to a threshold. The output then saturated at either the high voltage rail or the low voltage rail. For configurable level shifting, the logic level generator supplies the V+ and V- rails of the op-amp. For tri-state control, an analog switch is used on the output of the op-amp, allowing the op-amp to be disconnected and the I/O pin to float when in tristate mode.

The LM7171 op-amp was chosen for this application. It is rated for a maximum supply voltage of 36 (between V+ and V-) and has defined parameters when used with ±15V rails. It has a gain-bandwidth product of 200MHz and a maximum slew rate of 4100 V/us.

With a 1.65V differential input voltage (3.3V microcontroller output - 1.65V threshold), the op-amp can drive 10MHz signals at an amplitude of 33V before hitting the bandwidth limit:

$$1.65V \times 200MHz \cdot \frac{V}{V} \times \frac{1}{10MHz} = 33V$$

Considering only the slew rate limit, the op-amp can drive a 10MHz signal with 2ns rise/fall times at an amplitude of 8.2V before hitting the slew rate limit:

$$4100\frac{V}{us} \times 0.002ns = 8.2V$$

Typically high-speed and high-voltage operation is mutually exclusive due to signal integrity and the required drive currents. From the above analysis, the LM7171 should be able to drive high-speed, low-voltage signals (e.g. 0 to 5V @ 10MHz) or low-speed, high-voltage signals (e.g. -15V to +15V @ 10kHz) with no issue.

It is important to note that these ratings are maximum values, and the actual ratings vary with supply voltage, temperature, load, and many other factors. From the LM7171 datasheet, Figure 42 shows the GBWP and slew rate variation with supply voltage, and Figure 43 shows the variation with load capacitance.

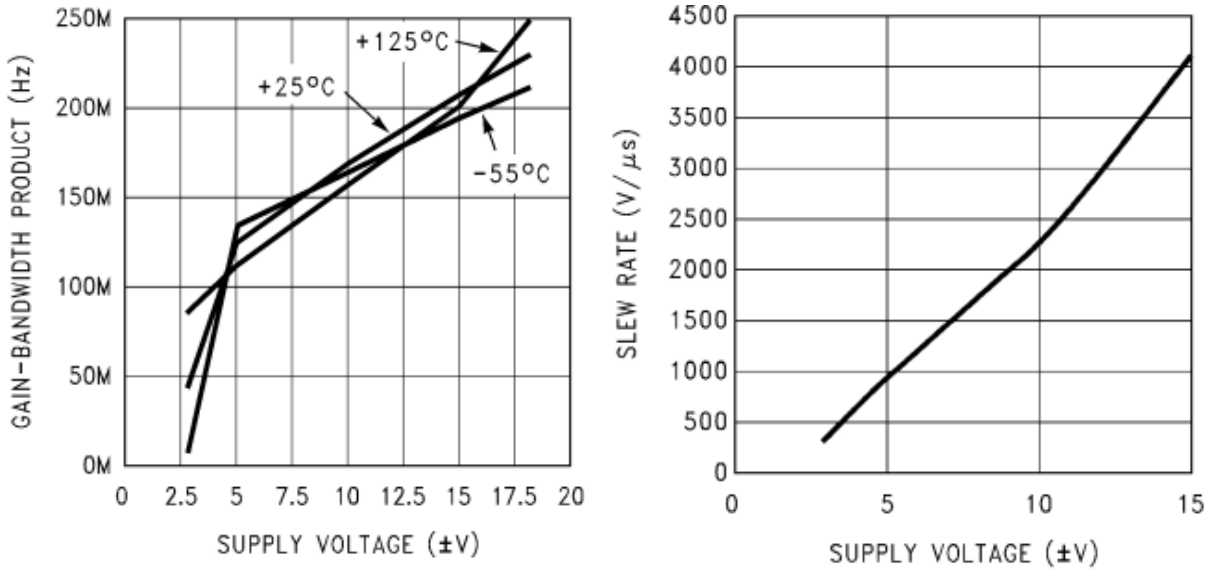


Figure 42: LM7171 Datasheet GBWP and Slew Rate Variation with Supply Voltage

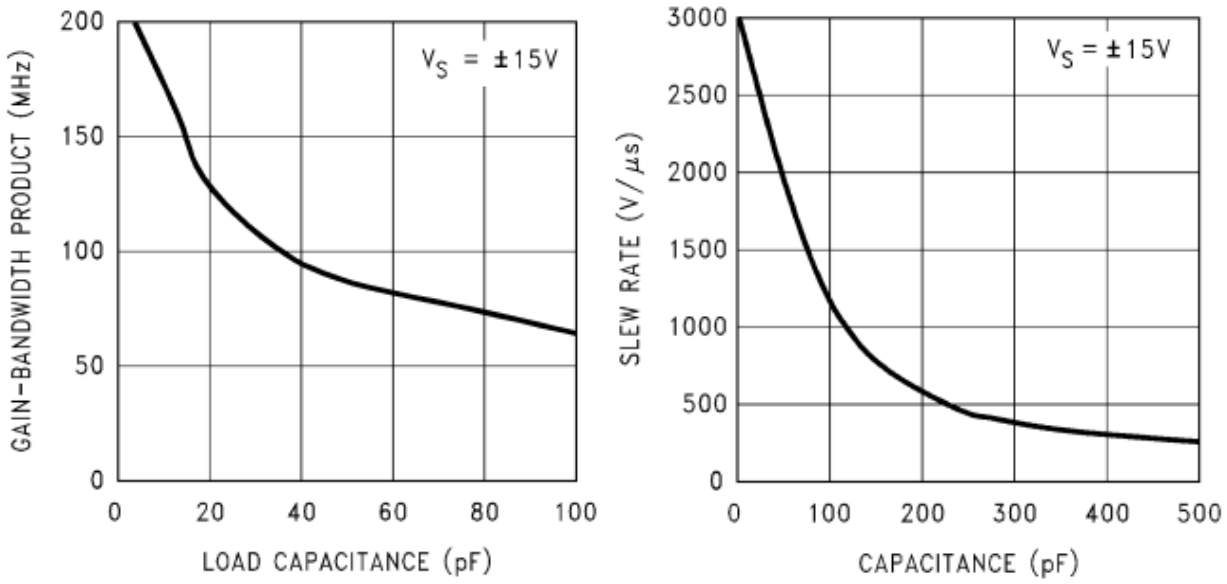


Figure 43: LM7171 Datasheet GBWP and Slew Rate Variation with Load Capacitance

Initial testing of the output driver showed promising results. Figure 44 shows the measured output waveform with V_H = 15V and V_L = 0V given a 0-3.3V 100kHz input from the microcontroller. Since the LM7171 is not a rail-to-rail op-amp, there is some voltage loss on the

output. With V_H = 15V, the actual high voltage of the output waveform is around 13.7V. This agrees with the datasheet specs for output voltage range.

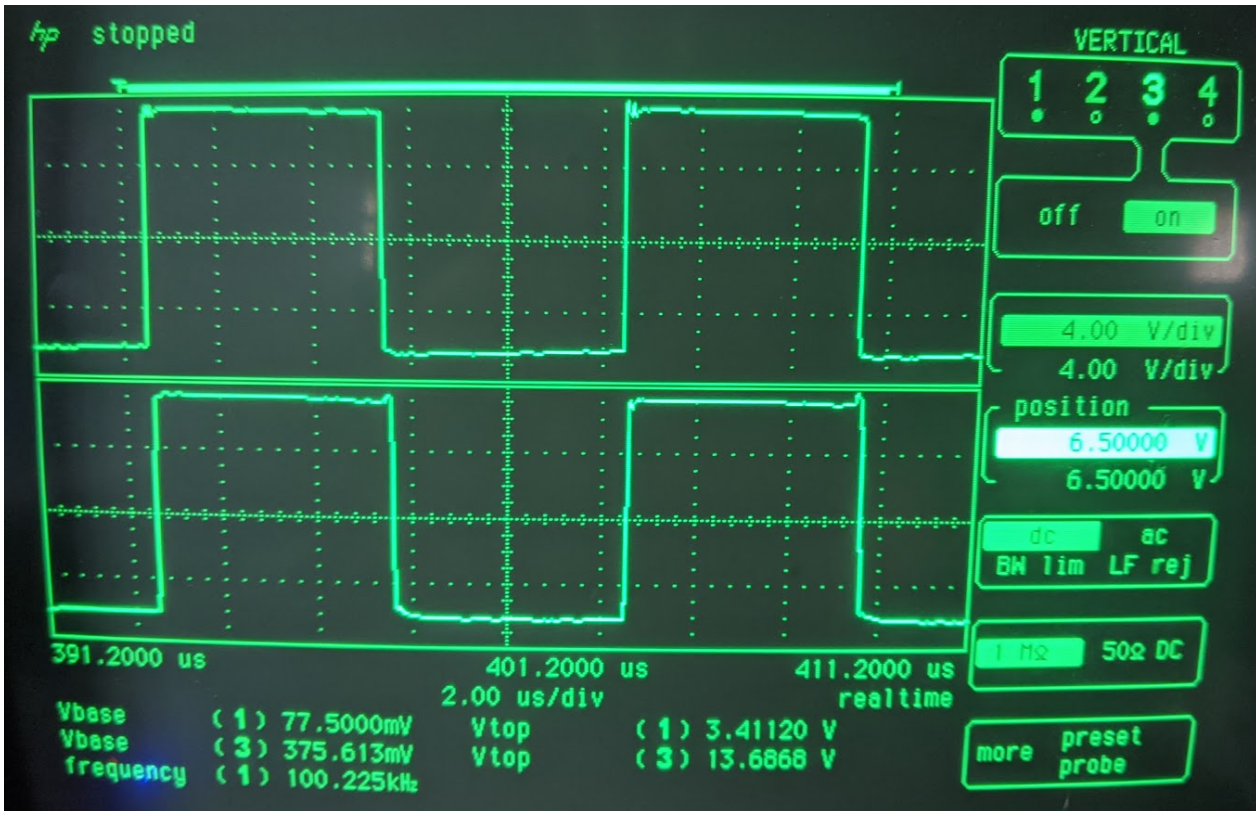


Figure 44: Output Driver Test Waveforms
(Top: 0-3.3V Input from Microcontroller, Bottom: 0-13.7V Scaled Output with V_H = 15V, V_L = 0V, f = 100kHz)

On some I/O pins however, the output waveform showed a very slow fall time. The waveform shows evidence of possibly some additional loading or capacitive coupling on the I/O pin. After further testing, the issue has been narrowed down to particular LM7171 chips and persists even when there is no load connected to the op-amp. This might be due to layout differences between the I/O pins or variation in parameters between chips. No further investigation has been done due to recent events. A future improvement might involve redesigning the output driver circuit for

closed-loop operation with a high enough gain to ensure that the output remains saturated, therefore improving consistency between I/O pins.

The single-ended receiver and differential receiver circuits were updated with the LM7171 op-amp and the feedback loops were redesigned for ±15V rails. The schematics are shown in Figures 45 and 46, respectively. At this time, only basic testing has been done on the single-ended receiver circuit. Differential signalling has not been tested.

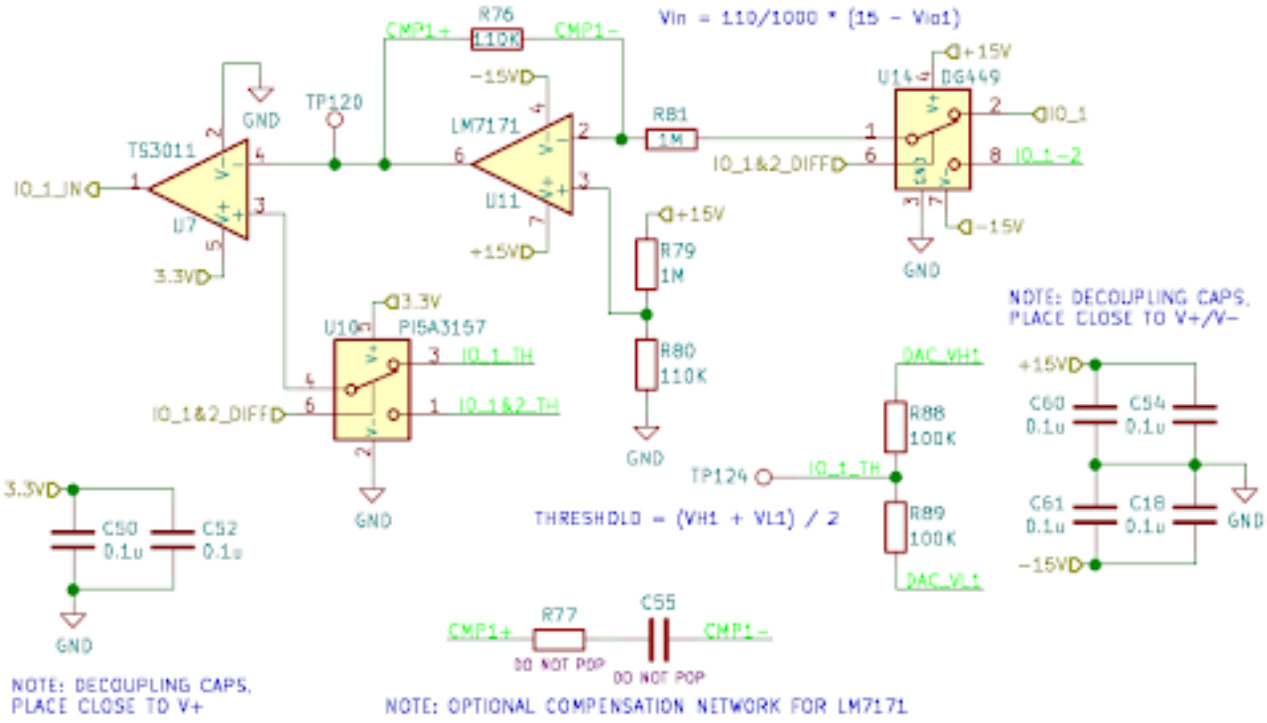


Figure 45: Single-Ended Receiver Schematic

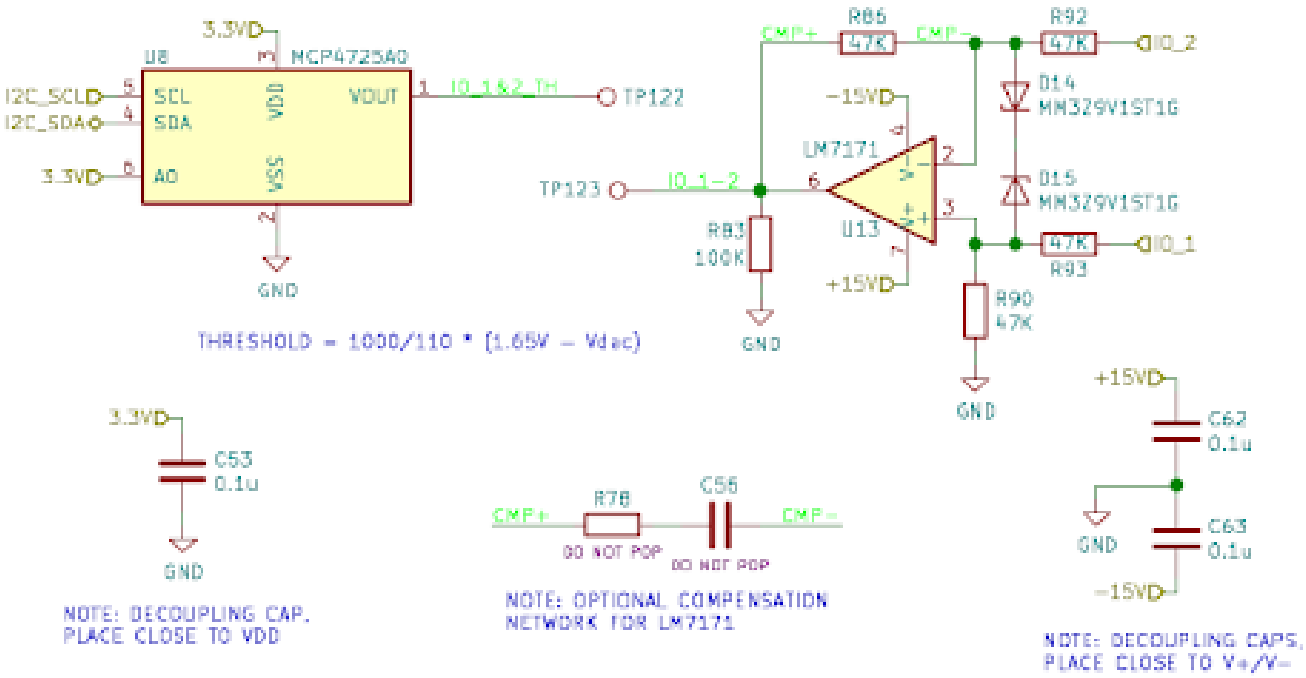


Figure 46: Differential Receiver Schematic

The configurable resistors portion of the level shifter was designed using common transistors to minimize cost. The resistors are only enabled or disabled during the board configuration stage, so no high-speed operation is necessary. Each pull-up or pull-down resistor on an I/O pin is driven by a combination of common emitter circuit and FET, shown in Figure 47. The FET provides very low on resistance when enabled, however its gate must be driven to either +15V or -15V (depending on whether its a pull-up or pull-down) to ensure that the transistor is fully on. This is accomplished using the common emitter circuit, which in turn is driven with a 3.3V signal from the microcontroller.

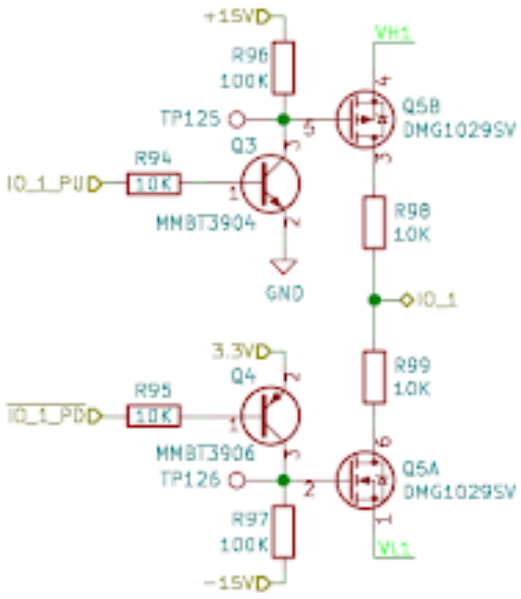


Figure 47: Configurable Pull-up/Pull-down Resistor Schematic

A similar setup is used for the termination resistors. A common emitter circuit drives a pair of back-to-back FETs, shown in Figure 48. The pair of FETs are required because current through the termination resistor may be bidirectional. A split termination scheme is used to reference the differential pair to the I/O Master ground. The 120Ω resistor used in non-split terminations is therefore divided into two series 60Ω resistors. Due to wattage requirements and since 120Ω is a common resistor size, each 60Ω termination resistor is implemented using a parallel combination of 1/8 watt 120Ω resistors. (IG)

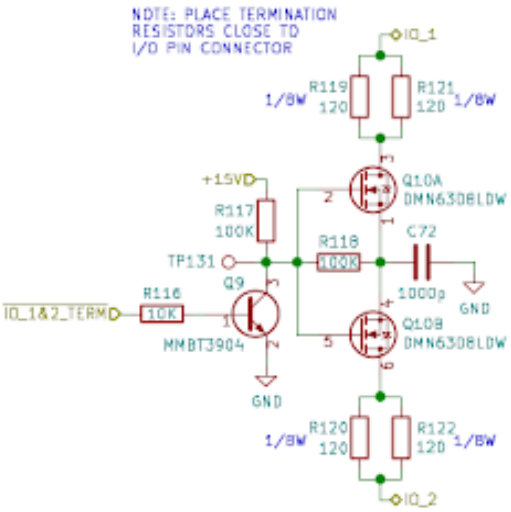


Figure 48: Configurable Termination Resistor Schematic

Figure 49 shows the complete PCB layout of the level shifter subsystem for a pair of I/O pins.

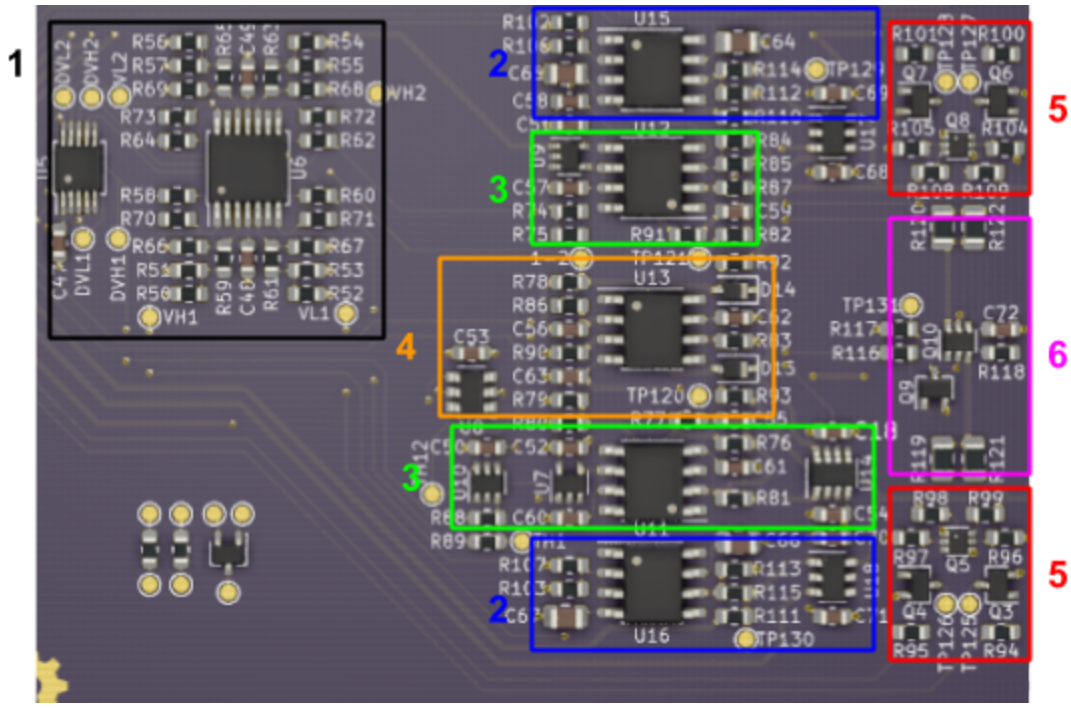


Figure 49: Level Shifter PCB Layout
(1 - Logic Level Generator, 2 - Output Driver, 3 - Single-Ended Receiver,
4 - Differential Receiver, 5 - Pull-up/Pull-down Resistors, 6 - Termination Resistor)

5.2.8 Circuit Protection

The over-voltage protection feature of the I/O Master is designed to handle over-voltages on the I/O pins that are outside of the ±15V rails. This is accomplished by taking advantage of zener diodes. The over-voltage protection circuit is shown in Figure 50.

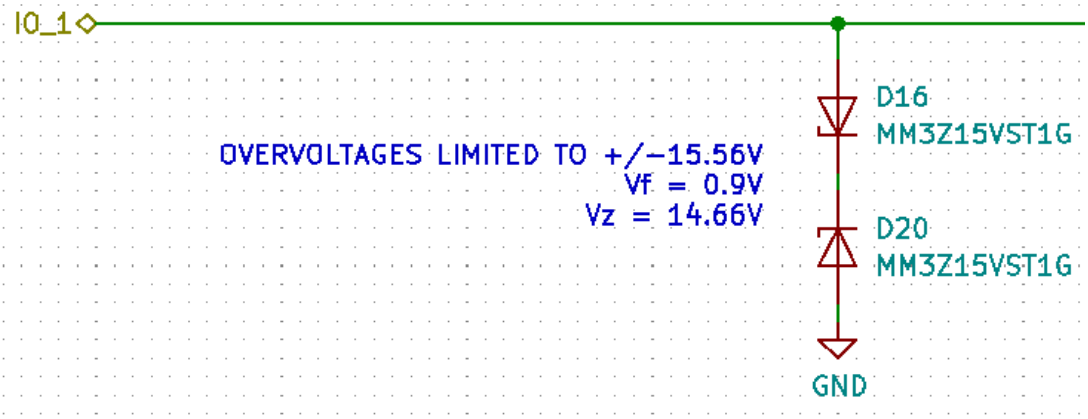


Figure 50: Over-voltage Protection Schematic

The zener breakdown voltage was chosen to be 14.66V, accounting for the forward voltage drop of 0.9V of the other diode. This over-voltage protection scheme clamps over-voltages to ±15.56V, which is sufficient in protecting any circuitry on the I/O Master that is only rated for ±15V operation. The over-current protection circuit helps limit the zener diode current during clamping.

The over-current protection circuit is designed to limit the current of each I/O pin to approximately 50mA. This protects the I/O Master from short circuits on the I/O pins or target device circuitry. The over-current protection schematic is represented in Figure 51.

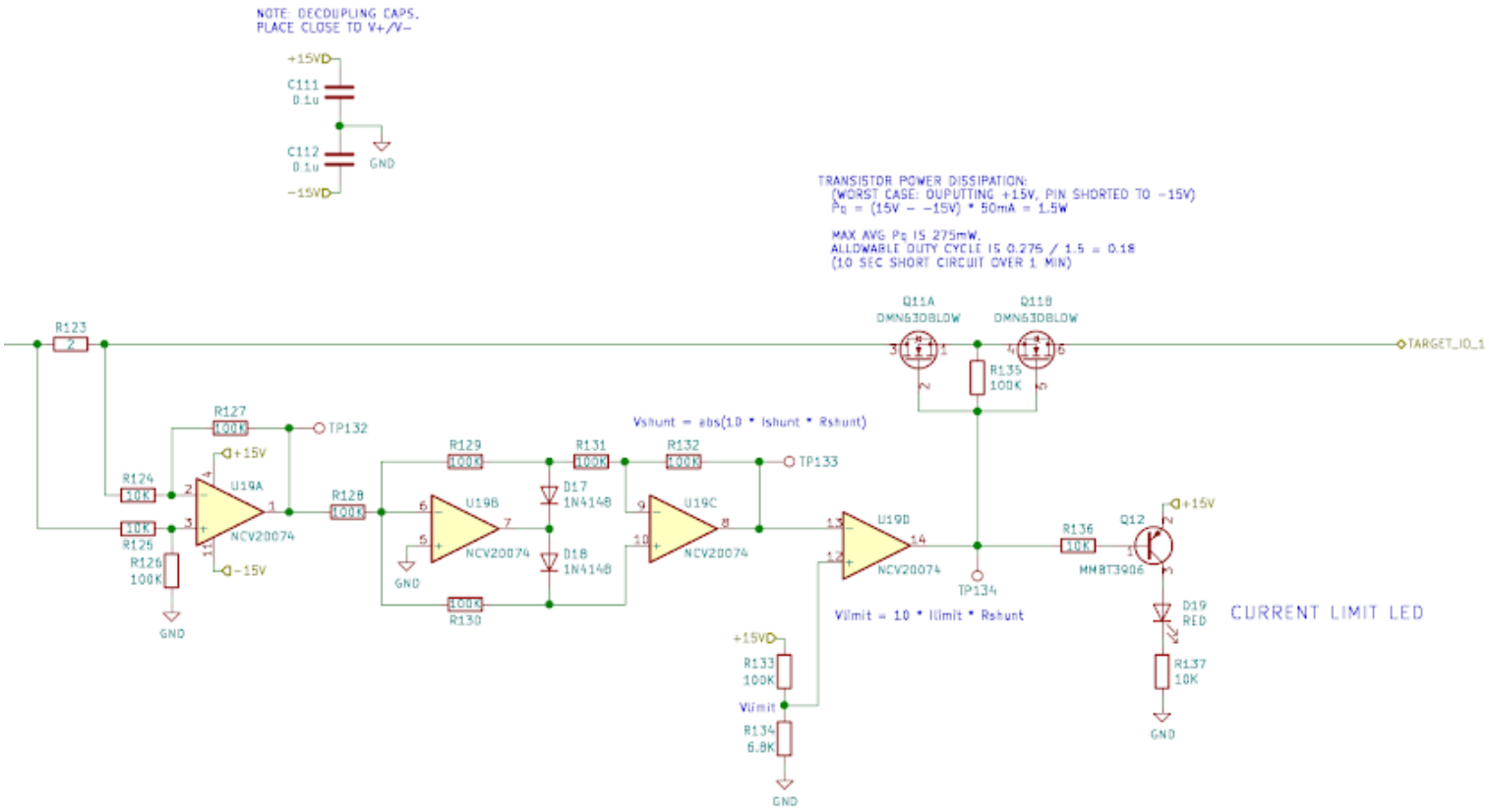


Figure 51: Over-current Protection Schematic

To reduce the number of unique parts on the I/O Master, the same low-speed quad op-amps from the level shifter were used to implement the feedback loop. A standard 1% 1/10W 0603 sense resistor is used to reduce cost, as current accuracy is not required for a protection circuit. At the maximum current limit of 50mA, the power dissipated by the sense resistor is 5mW, which is significantly under the 0603 package maximum power rating of 100mW. The current limiting FETs are driven by the last op-amp in the feedback loop, which is powered from the ±15V rails. The current limit setpoint is determined by a voltage divider (R133 and R134 in Figure 51) which should match the error amplifier's voltage output at maximum current. The maximum power dissipation of the FETs occurs when the over-current protection circuit is limiting current

through a dead short between the +15V and -15V rails. With 30V across the FETs and a maximum current of 50mA, this results in a total power dissipation of 1.5W. Note that the maximum power rating of the dual FET package is 275mW. As a result, the over-current protection circuit cannot handle continuous current limiting. At the maximum FET package dissipation of 275mW, current limiting could be duty cycled at 18%. This is equivalent to a 10 second short circuit in a 1 minute time frame. Operation beyond 10 seconds would result in overheating of the FETs. This downside could potentially be improved in a later revision through a redesign of the circuit.

When an over-current condition occurs, visual feedback is given to the user using a red LED. This allows the user to power off the I/O Master or correct the fault condition in order to prevent hardware damage. Figure 52 shows operation of the over-current protection circuit during testing. (NU)

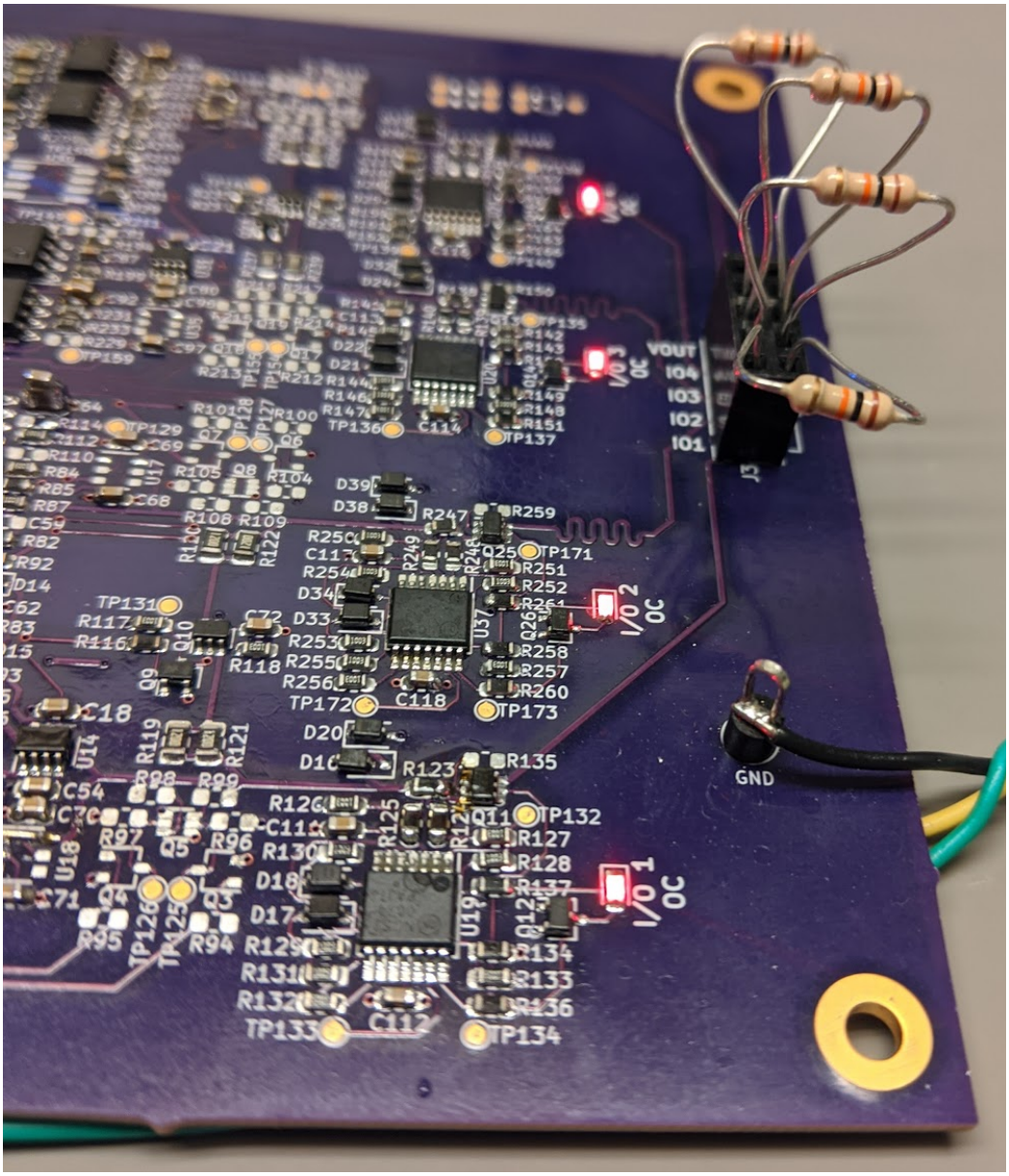


Figure 52: Over-current Protection Visual Indication

5.3 Software Design and Implementation

5.3.1 General Overview

The I/O Master software is split into two main components: the firmware that runs on the microcontroller, and the computer control software that is paired with it. The microcontroller

firmware will have support for USB communications, onboard electronics configuration for protocols, ability to read in and write out data at 10 MHz through DMA, and a scheduler to ensure all of these can run without the firmware locking up, crashing, or losing data. The computer control software is the user's way of interfacing with the I/O Master. It will feature the ability to set configurations, send and receive data, and give the user information on how to physically connect the target device to the I/O Master. Users could also write their own programs to control and use the I/O Master by reusing the functions written for the computer control software.

Figure 53 shows a level 0 block diagram for the software. Tables 38 and 39 list the functional requirements for the blocks shown in Figure 53.

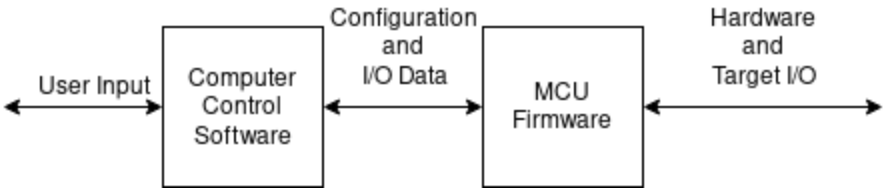


Figure 53: Software Level 0 Block Diagram

Table 38: Computer Control Software Functional Requirements

Module	Computer Control Software
Designer	Corey Dye
Inputs	User Inputs, I/O Data
Outputs	I/O Data, Configuration Data
Description	The Computer Control Software is the user's way of interacting with the I/O Master. From here, they will be able to configure voltage levels and frequencies for specified protocols. The user will also be able to send and receive data from the control software.

Table 39: MCU Firmware Functional Requirements

Module	MCU Firmware
Designer	Aaron Dubler, Cameron Sinko
Inputs	Configuration and I/O Data
Outputs	I/O Data and Hardware I/O
Description	The MCU Firmware is responsible for reading and writing data between the target device and the computer. This firmware will run on the I/O Master's on-board microcontroller. The firmware will also be responsible for board configuration to set voltage levels, frequencies, and pins for the user specified protocol.

This firmware must be highly optimized and efficient to ensure that data can always be sent in and out of the I/O master at the desired speed. To achieve this, a scheduling loop will be implemented that will make use of all clock cycles available on the I/O Master. The scheduler will ensure that queued data is being moved through the I/O Master before any of the buffers fill up. This will ensure that a 10MHz speed is achievable. To communicate with a computer, The I/O Master firmware will have support to send and receive data and commands over USB. When a configuration command is received, the device will prepare itself to send data using the settings specified in the command. The clock speed will be set by adjusting the MCU timers that synchronize data transmission. For each I/O pin that will be used, a DAC will be configured using I^2C to set the voltage high and voltage low for that pin. Each pin will also be configured to turn on/off pull-up resistor, turn on/off pull-down resistor, set single or differential mode, and (if using differential mode) turn on/off terminator resistor by sending control signals to connected pins on the MCU.

The computer control software will be a GUI implemented with a Python based program to create an intuitive, user friendly, visually appealing interface that requires little to no research to

use. The interface will feature drop down menus to select the desired protocols and voltage levels to be used on the device. Once a protocol is selected, the proper pins, voltage levels, and clock speed will be communicated to the board using USB 2.0 so that the microcontroller firmware can configure everything on-board. The GUI will also let the user send and receive data and have a text box to send commands and one to receive commands. (CS/AD/CD)

Figure 54 shows a level 1 block diagram for the software. Tables 40, 41, 42, and 43 list the functional requirements for the blocks shown in Figure 54.

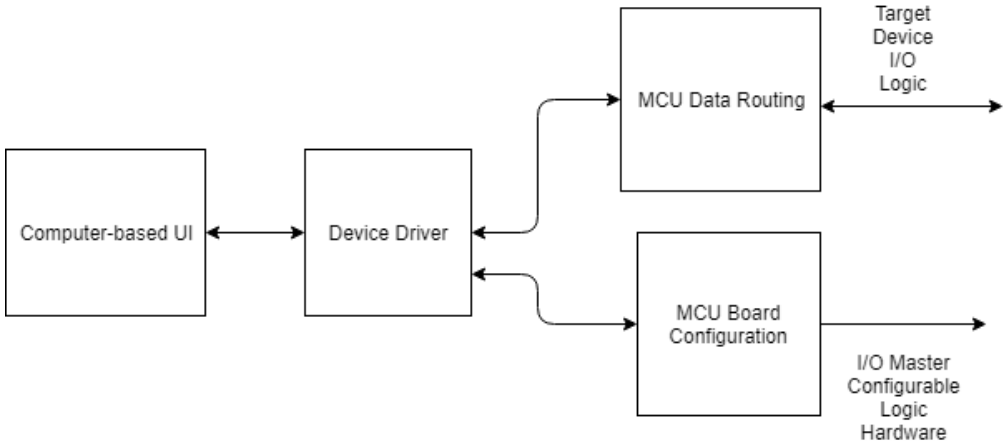


Figure 54: Software Level 1 Block Diagram

Table 40: Computer-based UI Functional Requirements

Module	Computer-based UI
Designer	Corey Dye
Inputs	User Data, Data from Device
Outputs	Data to Device
Description	The Computer based UI is the user interface to read and write data for interfacing with the I/O Master. It will feature dropdown menus, text inputs, and message boxes for user interaction.

Table 41: Device Driver Functional Requirements

Module	Device Driver
Designer	Corey Dye
Inputs	Data from UI, Data from Device
Outputs	Data to UI, Data to Device
Description	The Device Driver is the back end code used to send and receive data with the I/O Master. This data will work with the Computer-based UI to allow the user to interface with the I/O Master.

Table 42: MCU Data Routing Functional Requirements

Module	MCU Data Routing
Designer	Cameron Sinko
Inputs	Data from computer, Data from target device
Outputs	Data to computer, Data to target device
Description	The MCU Data Routing block is responsible for sending data between the target device and the computer. DMA will be utilized to send and receive data without utilizing CPU resources. The CPU will be responsible for scheduling data to send and process, along with data formatting.

Table 43: MCU Board Configuration Functional Requirements

Module	MCU Board Configuration
Designer	Aaron Dubler
Inputs	Configuration Commands
Outputs	Configuration I/0
Description	This module will take configuration commands and apply them to the device. This includes setting the routes for the data handler, setting the clock speed, designating input and output pins, and setting the logic voltage level.

The I/O Master goes through an initial configuration sequence before it is ready to send and receive data. The steps shown in Figure 55 utilize the MCU Board Configuration Block of the MCU firmware. (CS)

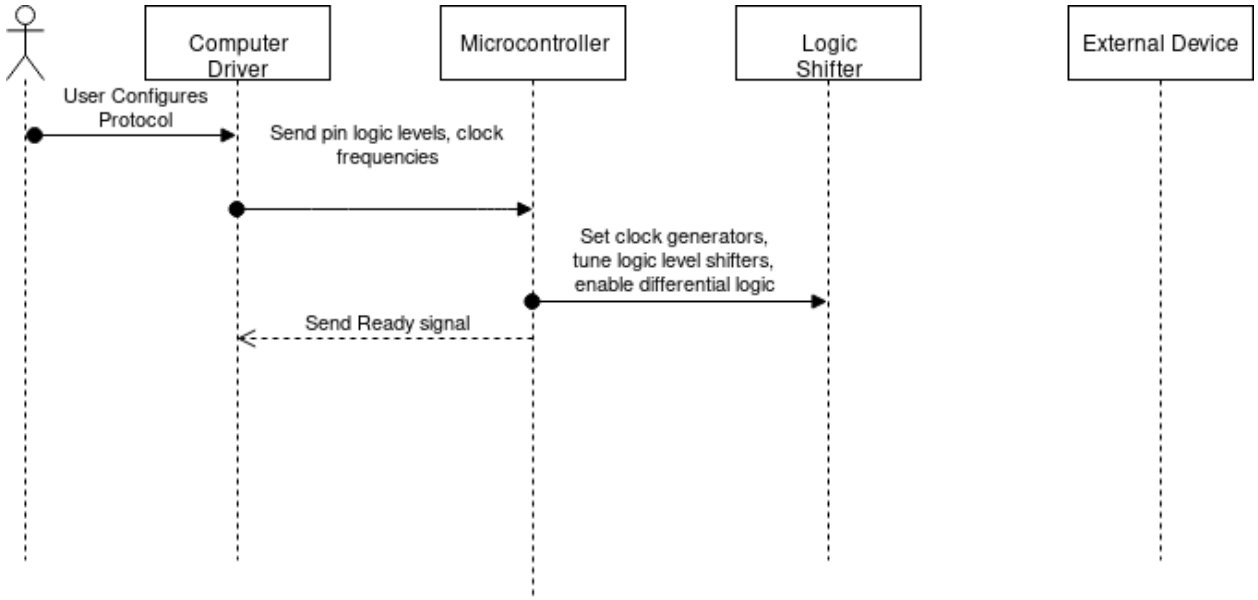


Figure 55: Sequence Diagram of I/O Master Initialization

First, the user will configure the software to the desired protocol. Then, the computer will send the corresponding logic levels, clock frequencies, and other configuration data to the microcontroller. The microcontroller will parse this data and configure the on-board hardware. Finally, a ready signal is sent back to the computer. (CS)

The steps to send data are shown in Figure 56.

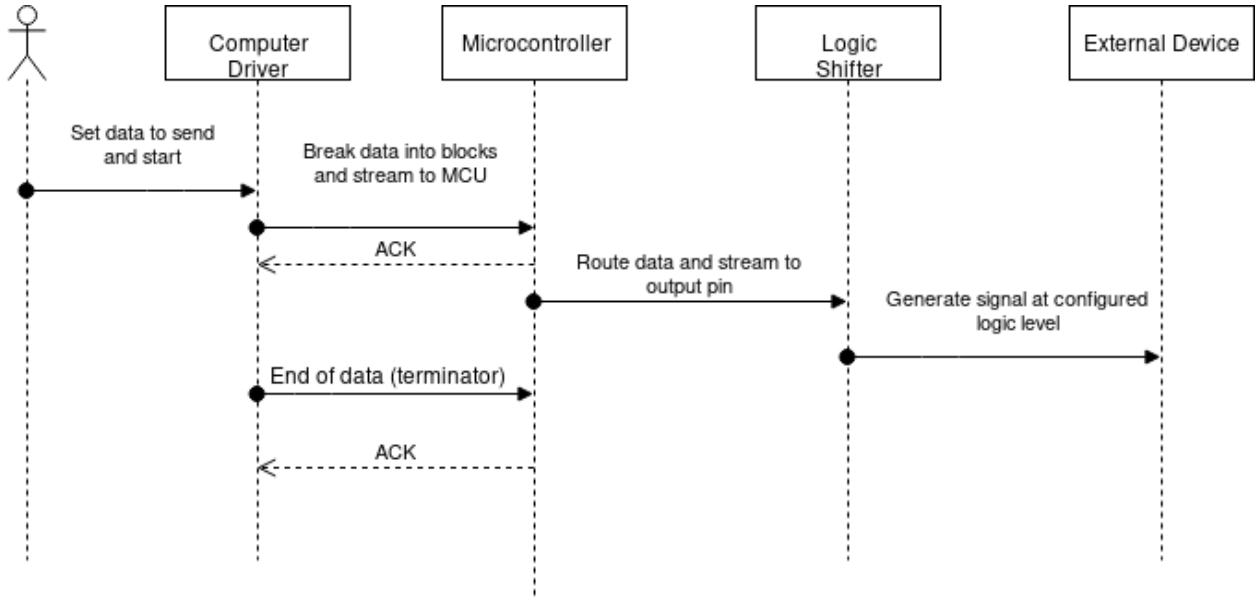


Figure 56: Sequence Diagram of Sending Data to an External Device

To send data, the user will specify what data to send. The computer software will break this data into chunks and stream them to the microcontroller. As the microcontroller receives and processes data, it will send an acknowledge command back to the computer software. The microcontroller will route the data to the configured output pin, to which it is then handled by on-board hardware. Once all of the data is sent, the process is automatically halted. (CS)

The steps to receive data are shown in Figure 57.

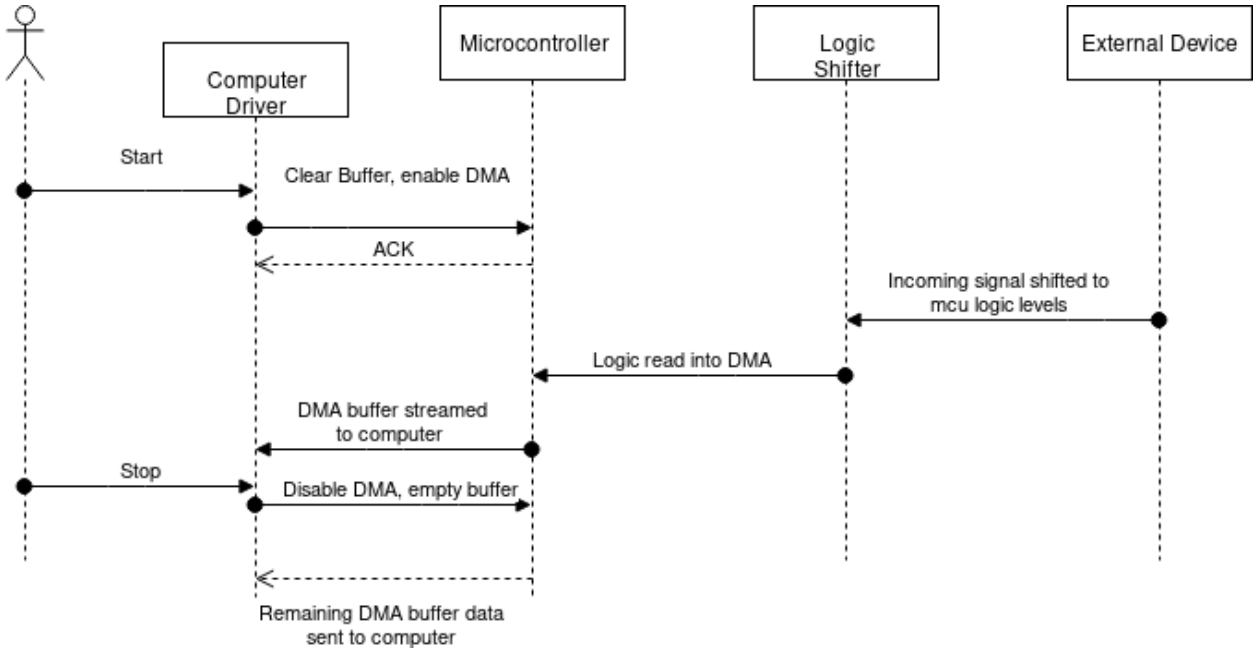


Figure 57: Sequence Diagram of Receiving Data from an External Device

To receive data, the user must start the I/O Master. A signal will be sent to the microcontroller and the onboard buffer will be cleared and DMA enabled. As logic is read into the microcontroller, it will be formed into packets and sent to the computer software. This process will continue unless the computer software explicitly stops the process. (CS)

Figure 58 shows a level 2 block diagram of the MCU firmware. Tables 44, 45, 46, 47, 48, and 49 list the functional requirements for the blocks shown in Figure 58.

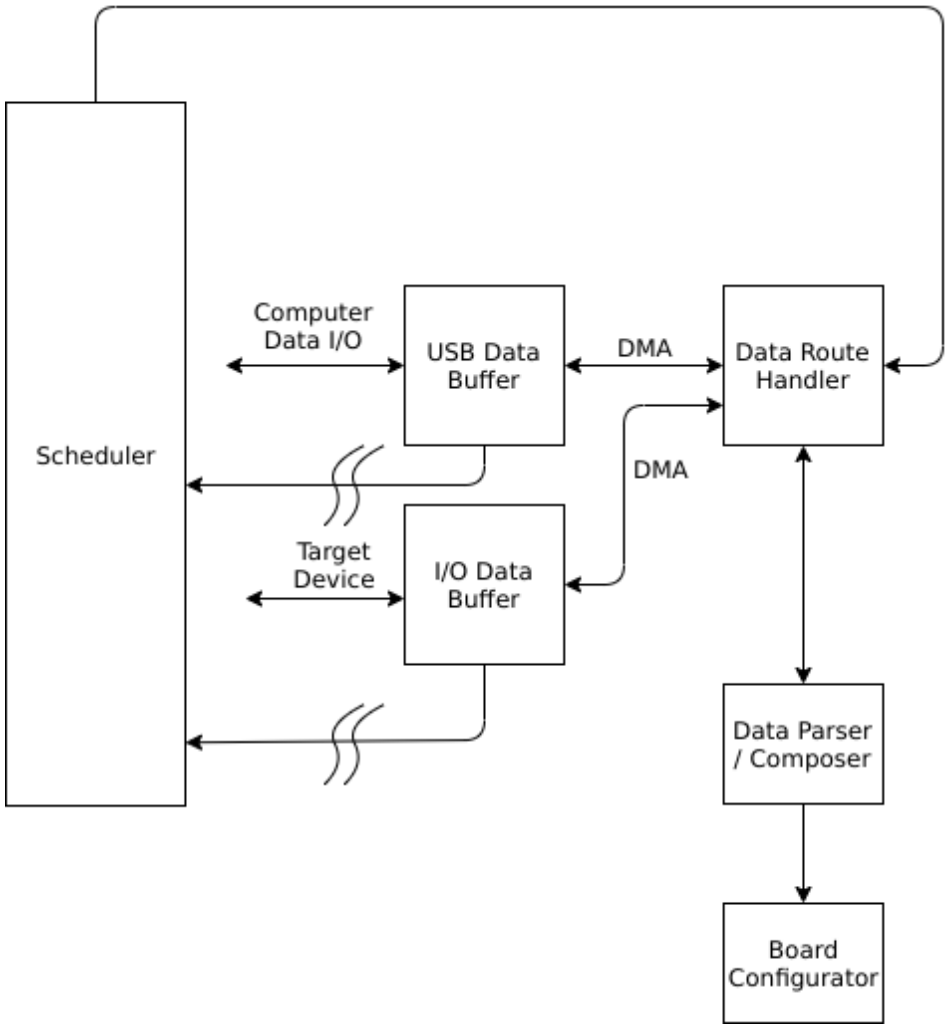


Figure 58: Microcontroller Firmware Level 2 Block Diagram

Table 44: Scheduler Functional Requirements

Module	Scheduler
Designer	Cameron Sinko
Inputs	Data Interrupts from USB and I/O to Target Device through Flags
Outputs	Data handle to the Data Route Handler
Description	This module is in charge of handling interrupt signals that occur when data enters the microcontroller and is the main running loop on the microcontroller. The scheduler will determine the priority of handling the data, and schedule when to call the data route handler. The determined task will be handled by the Data Route Handler.

Table 45: Data Route Handler Functional Requirements

Module	Data Route Handler
Designer	Cameron Sinko
Inputs	Data handle
Outputs	Data
Description	The Data Route Handler represents all of the processes to read in or send out data on the I/O Master. This module is called from the scheduler and from incoming data that is ready to be sent out immediately (overriding the scheduler). These processes will set up the microcontroller hardware to actually send the data and handle management of incoming and outgoing data queues.

Table 46: Data Parser/Composer Functional Requirements

Module	Data Parser/Composer
Designer	Cameron Sinko, Aaron Dubler
Inputs	Raw / Formatted Data
Outputs	Formatted / Raw Data
Description	This module is a data interpreter and converter. It can parse raw data and split into packets or frames, along with reading and interpreting data from computer command packets.

Table 47: Board Configurator Functional Requirements

Module	Board Configurator
Designer	Aaron Dubler
Inputs	Configuration commands
Outputs	ConfigurationI/0
Description	This module will take configuration commands and apply them to the device. This includes setting the routes for the data handler, setting the clock speed, designating input and output pins, and setting the logic voltage level.

Table 48: I/O Data Buffer Functional Requirements

Module	I/O Data Buffer
Designer	Cameron Sinko
Inputs	Data to send / Data to receive
Outputs	Sent data / Data received
Description	The I/O Data Buffer handles inputting and outputting data through GPIO. DMA is utilized to stream data between GPIO pins

Table 49: USB Data Buffer Functional Requirements

Module	USB Data Buffer
Designer	Cameron Sinko, Aaron Dubler
Inputs	USB Data to send / USB Data to receive
Outputs	Sent Data / Data Received
Description	The USB Data Buffer handles all communications between the I/O Master and the user computer. The Library implementations for the STM32 hardware USB will be utilized to do so.

5.3.2 Implementation Overview

The software implementation for the I/O Master utilizes the C programming language for the firmware and Python for the GUI. For a software project to be feasible to maintain and improve, the layout and design of the implementation are just as important as the actual design of the logic of the software. By segregating the code for the I/O Master logic from the functions to actually control the hardware, the software can be easy to read and understand without needing to know specifics about the hardware implementation and the surrounding frameworks.

The design of the MCU firmware is split into three parts. The first part contains all of the functions specific to starting and running code on the microcontroller, which is the *main.c* file and its supporting definitions that define very basic things like what frequency the

microcontroller is operating at. This code is generally automatically created when a new project is made for a microcontroller and is mostly unmodified except the additions to start the I/O Master software and main loop. The second part is the logical code implementation for the I/O Master. The command structure, data models for I/O pins and configurations, and the data structures and algorithms for reading and writing data are in this part. No matter what microcontroller the I/O Master were to use, this code can be reused without any changes needed. In addition, the functions for hardware actions, such as writing data to a DAC are defined in general purpose header files. The third part of the code is the implementation of the general purpose functions for the specific MCU being used, the STM32H743 in this case. This code utilizes the ST Hardware Abstraction Layer (HAL) library to handle common things like turning on and off pins and sending data through I2C lines. For more time sensitive tasks like starting output clocks and syncing them to data output, the MCU registers are set manually using bitwise operations such as AND, OR, and NOT.

With the firmware split into these three parts, there can be focus on simplifying the design of the logical implementation code, as that's the true code that will be written, debugged, and edited long term on the project. For the functional MCU hardware implementations, the code may be more complex and confusing, but once it is tested and working, the hard can just be utilized as easier to understand functions. (CS)

The implementation of logging is a good general example of how the software design looks like throughout the firmware. To log a message anywhere in the program after initialization, the following function can be used (shown right before the main loop starts):

```
IOMLog("I/O Master is Ready!\n");
```

This function is very simple to use, you can give it any string and it will be sent over the UART. *IOMLog* is one of the general use functions that is defined in the *log.h* header file.

```
///
/// @brief logs a message
///
/// @param msg the message to log
///
void IOMLog(const char* msg);
```

Here, the function prototype is clearly defined and documented. The *@brief* summary states that this function *logs a message* and it takes a *@param* of *msg, the message to log*. The *IOMLog* function could be implemented on any interface that is desired, but we chose to use a UART interface. in *src/log.c* lives the implementation:

```
void IOMLog(const char* msg) {
     UARTSendString(msg);
}
```

IOMLog simply sends the string over UART with the *UARTSendString* Hardware implementation function.

```
IOM_ERROR UARTSendString(const char* string) {
     uint16_t size = 0;
     while (string[size] != '\0') size++;
     uint8_t* pData = malloc(size);
     memcpy(pData, string, size);
     return UARTQueueTXData(pData, size);
}
```

The string is allocated into the memory heap, then it is added to the UART TX Data Queue, which is later explained. In addition *UARTSendString* can return an error type called *IOM_ERROR*. This error type is commonly used throughout the program, but is ignored for the

logging case as logging is a best effort attempt. If logging were to fail, it does not matter because it is a low priority task. As shown, this leads down a series of other functions beyond *UARTQueueTXData* before the data actually is sent through the hardware. For actual development, it does not matter how this works under the hood after it is written, so it is wrapped within the *IOMLog* function to create an easy to understand method of logging data. Modern compilers are able to optimize the machine code that is generated, so the benefits of structuring code in a more readable format only come at the cost of the time spent designing it in the first place.

The majority of the code for the I/O Master firmware follows this same structure. To make it easiest to understand in this writing, the higher level logical code is first explained, then the lower level code that controls the hardware will come afterwards for each component. (CS)

5.3.3 State Machine

While not necessary to implement the design, a state machine being utilized in the MCU firmware makes it much easier and safer to ensure that the MCU does not get locked up or crash on accident. The state machine has 5 states: Init, Config, Ready, Busy, and Error, shown in Figure 59.

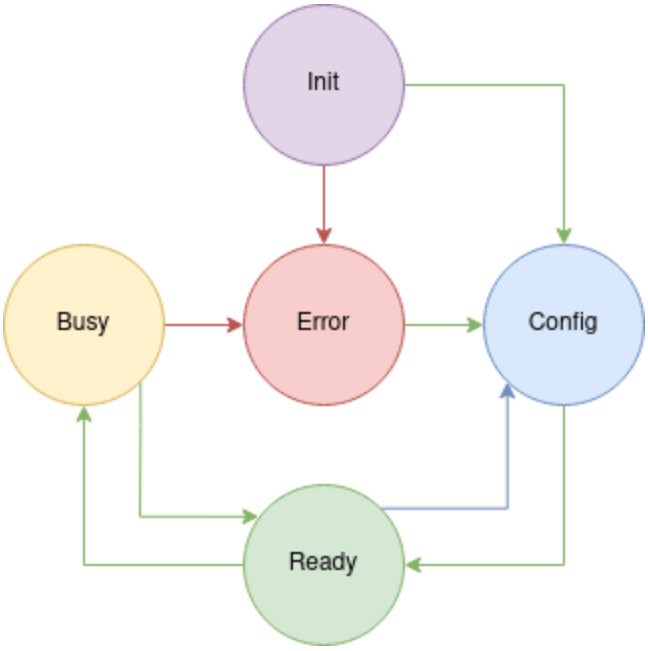


Figure 59: State Machine Implementation for MCU Firmware

The I/O Master begins in the Init state. In this state, the MCU initializes the hardware. If there is an error, it will go to the error state and halt in this state. If initialization is successful, the device enters the Config state. All of the I/O Master board hardware can be configured from the Config state by sending configuration commands. Once the board is configured, a command will be sent to let the I/O Master enter the Ready state. Here, data can be loaded in to be sent out. Once enabled, the device will enter the Busy state and data I/O will commence. If there are any errors that occur, the device will enter the Error state, which may or may not be recoverable back to the Config state, depending on the error. If the device successfully reads/writes data, it returns back to the ready state.

This state machine ensures that hardware must be initialized before configuring it, along with making sure the device is explicitly changed to the Ready state before it is actually used to send

and receive data. If it were possible to directly start sending data right after initialization, there would be an increased risk in damaging the user's connected device.

In code, the state machine is implemented as an enumeration. The actual state is tracked with the *IOMState* variable. Throughout the software, this state is updated and enforced. (CS)

```
typedef enum {
IOM_STATE_READY, ///< Ready to send or receive data
IOM_STATE_INIT, ///< Device Initialization
IOM_STATE_CONF, ///< Device Configuration
IOM_STATE_BUSY, ///< Actively sending or receiving data
IOM_STATE_ERROR, ///< There was an error
} IOM_STATE;

IOM_STATE IOMState;
```

5.3.4 Initialization

Upon powering up the IO master the MCU runs many lines of initialization code to prepare itself for use. The first function of note is HAL_Init(). This function is included in the HAL Library and used to initialize the rest of the HAL Library for use.

__HAL_RCC_SYSCFG_CLK_ENABLE() enables the High Speed APB (APB2) peripheral clock. This is important for enabling certain GPIO pins. InitSystemClock() initializes the internal oscillator. This is used with the timers for synchronizing all components of data transmission including DMA and clock lines. (AD)

```
IOM_ERROR InitSystemClock(void) {
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
  HAL_PWREx_ConfigSupply(PWR_LDO_SUPPLY);
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE0);
  while(!__HAL_PWR_GET_FLAG(PWR_FLAG_VOSRDY)) {}

RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI | RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_BYPASS;
  RCC_OscInitStruct.HSIState = RCC_HSI_DIV1;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
```

```
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;

  RCC_OscInitStruct.PLL.PLLM = 8;
  RCC_OscInitStruct.PLL.PLLN = 160;
  RCC_OscInitStruct.PLL.PLLFRACN = 0;
  RCC_OscInitStruct.PLL.PLLP = 2;
  RCC_OscInitStruct.PLL.PLLR = 2;
  RCC_OscInitStruct.PLL.PLLQ = 4;

  RCC_OscInitStruct.PLL.PLLVCOSEL = RCC_PLL1VCOWIDE;
  RCC_OscInitStruct.PLL.PLLRGE = RCC_PLL1VCIRANGE_2;
  HAL_RCC_OscConfig(&RCC_OscInitStruct);

  RCC_ClkInitStruct.ClockType = (RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_HCLK |
RCC_CLOCKTYPE_D1PCLK1 | RCC_CLOCKTYPE_PCLK1 |
                                 RCC_CLOCKTYPE_PCLK2  | RCC_CLOCKTYPE_D3PCLK1);

  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.SYSCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB3CLKDivider = RCC_APB3_DIV2;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_APB1_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_APB2_DIV2;
  RCC_ClkInitStruct.APB4CLKDivider = RCC_APB4_DIV2;
  HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4);

  RCC_PeriphCLKInitTypeDef PeriphClkInitStruct = {0};

  PeriphClkInitStruct.PeriphClockSelection =
RCC_PERIPHCLK_USART3|RCC_PERIPHCLK_I2C3;
  PeriphClkInitStruct.Usart234578ClockSelection = RCC_USART234578CLKSOURCE_D2PCLK1;
  PeriphClkInitStruct.I2c123ClockSelection = RCC_I2C123CLKSOURCE_HSI;
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct) != HAL_OK)
  {
   return IOM_ERROR_INTERFACE;
  }
  return IOM_OK;
}
```

InitGPIO() initializes all pins to their default functions. This is done using the many definitions found in *mcu_conf.h. mcu_conf.h* contains many defines to make the rest of the code more easily read and understood within the rest of the code. Most notably the GPIO ports and pin names are given new definitions with more descriptive names. (AD)

```
IOM_ERROR InitGPIO(void) {
  GPIO_InitTypeDef GPIO_InitStruct = {0};
```

```
//Enable GPIO Port Clocks
__HAL_RCC_GPIOA_CLK_ENABLE();
__HAL_RCC_GPIOB_CLK_ENABLE();
__HAL_RCC_GPIOC_CLK_ENABLE();
__HAL_RCC_GPIOD_CLK_ENABLE();
__HAL_RCC_GPIOE_CLK_ENABLE();
__HAL_RCC_GPIOF_CLK_ENABLE();
__HAL_RCC_GPIOG_CLK_ENABLE();
__HAL_RCC_GPIOH_CLK_ENABLE();

//Determine what pins should be set to High (Active Low).
//All other pins will be set to Low (Active High)
const uint16_t GPIOA_SET_PINS = 0
;

const uint16_t GPIOB_SET_PINS = 0
| IO_1_PD_N_Pin
#ifdef IOM_8_IO_PINS
| IO_6_PD_N_Pin
#endif
| nRESET_Pin
;

const uint16_t GPIOC_SET_PINS = 0
| IO_3_4_TERM_N_Pin
| IO_4_PD_N_Pin
#ifdef IOM_8_IO_PINS
| IO_5_6_TERM_N_Pin
| IO_7_8_TERM_N_Pin
| IO_7_PD_N_Pin
| IO_8_PD_N_Pin
#endif
;

const uint16_t GPIOD_SET_PINS = 0
| IO_1_TRIS_N_Pin
| IO_2_TRIS_N_Pin
| IO_3_TRIS_N_Pin
| IO_4_TRIS_N_Pin
#ifdef IOM_8_IO_PINS
| IO_5_TRIS_N_Pin
| IO_6_TRIS_N_Pin
| IO_7_TRIS_N_Pin
| IO_8_TRIS_N_Pin
#endif
;

const uint16_t GPIOE_SET_PINS = 0
;

const uint16_t GPIOF_SET_PINS = 0
;
```

```
  const uint16_t GPIOG_SET_PINS = 0
  | IO_2_PD_N_Pin
  | IO_3_PD_N_Pin
  | IO_1_2_TERM_N_Pin
  #ifdef IOM_8_IO_PINS
  | IO_5_PD_N_Pin
  #endif
  ;

  const uint16_t GPIOH_SET_PINS = 0
  ;

//TODO determine if this pragma is the only way to avoid
// the overflow error for GPIOB set RESET pins
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Woverflow"
  //Write all of the pin states
  HAL_GPIO_WritePin(GPIOA, GPIOA_SET_PINS, GPIO_PIN_SET);
  HAL_GPIO_WritePin(GPIOA, ~GPIOA_SET_PINS, GPIO_PIN_RESET);

  HAL_GPIO_WritePin(GPIOB, GPIOB_SET_PINS, GPIO_PIN_SET);

  HAL_GPIO_WritePin(GPIOB, ~GPIOB_SET_PINS, GPIO_PIN_RESET);

  HAL_GPIO_WritePin(GPIOC, GPIOC_SET_PINS, GPIO_PIN_SET);
  HAL_GPIO_WritePin(GPIOC, ~GPIOC_SET_PINS, GPIO_PIN_RESET);

  HAL_GPIO_WritePin(GPIOD, GPIOD_SET_PINS, GPIO_PIN_SET);
  HAL_GPIO_WritePin(GPIOD, ~GPIOD_SET_PINS, GPIO_PIN_RESET);

  HAL_GPIO_WritePin(GPIOE, GPIOE_SET_PINS, GPIO_PIN_SET);
  HAL_GPIO_WritePin(GPIOE, ~GPIOE_SET_PINS, GPIO_PIN_RESET);

  HAL_GPIO_WritePin(GPIOF, GPIOF_SET_PINS, GPIO_PIN_SET);
  HAL_GPIO_WritePin(GPIOF, ~GPIOF_SET_PINS, GPIO_PIN_RESET);

  HAL_GPIO_WritePin(GPIOG, GPIOG_SET_PINS, GPIO_PIN_SET);
  HAL_GPIO_WritePin(GPIOG, ~GPIOG_SET_PINS, GPIO_PIN_RESET);

  HAL_GPIO_WritePin(GPIOH, GPIOH_SET_PINS, GPIO_PIN_SET);
  HAL_GPIO_WritePin(GPIOH, ~GPIOH_SET_PINS, GPIO_PIN_RESET);
#pragma GCC diagnostic pop

  //Default input pin configuration (Clock and I/O In/Out Pins)
  const uint16_t GPIOA_INPUT_PINS = 0
  | IO_1_CLK_Pin
  | IO_2_CLK_Pin
  | IO_3_CLK_Pin
  #ifdef IOM_8_IO_PINS
  | IO_8_CLK_Pin
  #endif
  ;
```

```
  const uint16_t GPIOB_INPUT_PINS = 0
  #ifdef IOM_8_IO_PINS
  | IO_5_CLK_Pin
  | IO_6_CLK_Pin
  | IO_7_CLK_Pin
  #endif
  ;

  const uint16_t GPIOC_INPUT_PINS = 0
  | IO_4_CLK_Pin
  ;

  const uint16_t GPIOD_INPUT_PINS = 0
  | IO_1_OUT_Pin
  | IO_2_OUT_Pin
  | IO_3_OUT_Pin
  | IO_4_OUT_Pin
  #ifdef IOM_8_IO_PINS
  | IO_5_OUT_Pin
  | IO_6_OUT_Pin
  | IO_7_OUT_Pin
  | IO_8_OUT_Pin
  #endif
  ;

  const uint16_t GPIOE_INPUT_PINS = 0
  | IO_1_IN_Pin
  | IO_2_IN_Pin
  | IO_3_IN_Pin
  | IO_4_IN_Pin
  #ifdef IOM_8_IO_PINS
  | IO_5_IN_Pin
  | IO_6_IN_Pin
  | IO_7_IN_Pin
  | IO_8_IN_Pin
  #endif
  ;

/**
  const uint16_t GPIOF_INPUT_PINS = 0
  //Must Configure Inputs
  ;

  const uint16_t GPIOG_INPUT_PINS = 0
  //Must Configure Inputs
  ;

  const uint16_t GPIOH_INPUT_PINS = 0
  //Must Configure Inputs
  ;

  **/
```

```
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;

GPIO_InitStruct.Pin = GPIOA_INPUT_PINS;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

GPIO_InitStruct.Pin = GPIOB_INPUT_PINS;
HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

GPIO_InitStruct.Pin = GPIOC_INPUT_PINS;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

GPIO_InitStruct.Pin = GPIOD_INPUT_PINS;
HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);

GPIO_InitStruct.Pin = GPIOE_INPUT_PINS;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);

//Must Configure GPIOF Inputs
//Must Configure GPIOG Inputs
//Must Configure GPIOH Inputs



//Special Pin Configurations

//ULPI Interface pin configuration
GPIO_InitStruct.Pin = 0
| ULPI_D0_Pin
| ULPI_CK_Pin
;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
GPIO_InitStruct.Alternate = GPIO_AF10_OTG2_HS;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

GPIO_InitStruct.Pin = 0
| ULPI_D1_Pin
| ULPI_D2_Pin
| ULPI_D3_Pin
| ULPI_D4_Pin
| ULPI_D5_Pin
| ULPI_D6_Pin
| ULPI_D7_Pin
;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
GPIO_InitStruct.Alternate = GPIO_AF10_OTG2_HS;
HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

```
  GPIO_InitStruct.Pin = 0
  | ULPI_STP_Pin
  | ULPI_DIR_Pin
  | ULPI_NXT_Pin
  ;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
  GPIO_InitStruct.Alternate = GPIO_AF10_OTG2_HS;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);


  //I2C Interface pin configuration
  GPIO_InitStruct.Pin = I2C_SDA_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_OD;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
  GPIO_InitStruct.Alternate = GPIO_AF4_I2C3;
  HAL_GPIO_Init(I2C_SDA_GPIO_Port, &GPIO_InitStruct);

  GPIO_InitStruct.Pin = I2C_SCL_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_OD;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
  GPIO_InitStruct.Alternate = GPIO_AF4_I2C3;
  HAL_GPIO_Init(I2C_SCL_GPIO_Port, &GPIO_InitStruct);

  //UART Interface pin configuration
  GPIO_InitStruct.Pin = 0
  | UART_TXO_Pin
  | UART_RXI_Pin
  ;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  GPIO_InitStruct.Alternate = GPIO_AF7_USART1;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  //USB Interface pin configuration
  GPIO_InitStruct.Pin = 0
  | USB_FS_DM_Pin
  | USB_FS_DP_Pin
  ;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  GPIO_InitStruct.Alternate = GPIO_AF10_OTG1_FS;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

/**
  const uint16_t GPIOA_AF_PINS = 0
  | ULPI_D0_Pin
```

```
    | ULPI_CK_Pin
    | I2C_SCL_Pin
    | UART_TXO_Pin
    | UART_RXI_Pin
    | USB_FS_DM_Pin
    | USB_FS_DP_Pin
    ;

    const uint16_t GPIOB_AF_PINS = 0
    | ULPI_D1_Pin
    | ULPI_D2_Pin
    | ULPI_D3_Pin
    | ULPI_D4_Pin
    | ULPI_D5_Pin
    | ULPI_D6_Pin
    | ULPI_D7_Pin
    ;

    const uint16_t GPIOC_AF_PINS = 0
    | ULPI_STP_Pin
    | ULPI_DIR_Pin
    | ULPI_NXT_Pin
    | I2C_SDA_Pin
    ;

    const uint16_t GPIOD_AF_PINS = 0
    ;

    const uint16_t GPIOE_AF_PINS = 0
    ;

    const uint16_t GPIOF_AF_PINS = 0
    ;

    const uint16_t GPIOG_AF_PINS = 0
    ;

    const uint16_t GPIOH_AF_PINS = 0
    ;
    **/


    //Low speed Output pins

    const uint16_t GPIOA_OUTPUT_PINS = 0
    #ifdef IOM_8_IO_PINS
    | IO_6_PU_Pin
    #endif
    ;

    const uint16_t GPIOB_OUTPUT_PINS = 0
    | IO_1_PU_Pin
    | IO_1_PD_N_Pin
```

```
| IO_4_PU_Pin
#ifdef IOM_8_IO_PINS
| IO_6_PD_N_Pin
#endif
;

const uint16_t GPIOC_OUTPUT_PINS = 0
| IO_4_PD_N_Pin
| IO_3_4_DIFF_Pin
| IO_3_4_TERM_N_Pin
#ifdef IOM_8_IO_PINS
| IO_5_6_DIFF_Pin
| IO_7_PU_Pin
| IO_7_PD_N_Pin
| IO_8_PU_Pin
| IO_8_PD_N_Pin
| IO_7_8_DIFF_Pin
| IO_5_6_TERM_N_Pin
| IO_7_8_TERM_N_Pin
#endif
;

const uint16_t GPIOD_OUTPUT_PINS = 0
| IO_1_TRIS_N_Pin
| IO_2_TRIS_N_Pin
| IO_3_TRIS_N_Pin
| IO_4_TRIS_N_Pin
#ifdef IOM_8_IO_PINS
| IO_5_TRIS_N_Pin
| IO_6_TRIS_N_Pin
| IO_7_TRIS_N_Pin
| IO_8_TRIS_N_Pin
#endif
;

const uint16_t GPIOE_OUTPUT_PINS = 0
;

const uint16_t GPIOF_OUTPUT_PINS = 0
| STATUS_R_Pin
| STATUS_G_Pin
| STATUS_B_Pin
;

const uint16_t GPIOG_OUTPUT_PINS = 0
| IO_2_PU_Pin
| IO_2_PD_N_Pin
| IO_1_2_DIFF_Pin
| IO_1_2_TERM_N_Pin
| IO_3_PU_Pin
| IO_3_PD_N_Pin
#ifdef IOM_8_IO_PINS
| IO_5_PU_Pin
```

```
  | IO_5_PD_N_Pin
  #endif
  ;

  const uint16_t GPIOH_OUTPUT_PINS = 0
  ;

  //Configure everything else as a low speed output pin
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;

  GPIO_InitStruct.Pin = GPIOA_OUTPUT_PINS;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  GPIO_InitStruct.Pin = GPIOB_OUTPUT_PINS;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  GPIO_InitStruct.Pin = GPIOC_OUTPUT_PINS;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  GPIO_InitStruct.Pin = GPIOD_OUTPUT_PINS;
  HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);

  GPIO_InitStruct.Pin = GPIOE_OUTPUT_PINS;
  HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);

  GPIO_InitStruct.Pin = GPIOF_OUTPUT_PINS;
  HAL_GPIO_Init(GPIOF, &GPIO_InitStruct);

  GPIO_InitStruct.Pin = GPIOG_OUTPUT_PINS;
  HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);

  GPIO_InitStruct.Pin = GPIOH_OUTPUT_PINS;
  HAL_GPIO_Init(GPIOH, &GPIO_InitStruct);

  return IOM_OK;
}
```

InitDAC() initializes the internal MCU DAC for setting the target device voltage.

```
IOM_ERROR InitDAC(void)
  {
    DAC_ChannelConfTypeDef sConfig = {0};

    /** DAC Initialization
    */
    hdac1.Instance = DAC1;
    if (HAL_DAC_Init(&hdac1) != HAL_OK)
    {
      return IOM_ERROR_INVALID; //TODO put a better error here
```

```
    }
    /** DAC channel OUT1 config
    */
    sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
    sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
    sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
    sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_DISABLE;
    sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
    if (HAL_DAC_ConfigChannel(&hdac1, &sConfig, DAC_CHANNEL_1) != HAL_OK)
    {
      return IOM_ERROR_INTERFACE;
    }
    return IOM_OK;
  }
```

InitI2CDAC() initialises I2C so it is ready for use in communication with the external DACs.

```
IOM_ERROR InitI2CDAC(void) {
    __HAL_RCC_I2C3_CLK_ENABLE();
    //TODO confirm this configuration is correct
    hi2c3.Instance              = I2C3;
    hi2c3.Init.Timing           = 0x10707EBE;
    hi2c3.Init.OwnAddress1      = 0; //TODO populate me with the correct value
    hi2c3.Init.AddressingMode   = I2C_ADDRESSINGMODE_7BIT;
    hi2c3.Init.DualAddressMode  = I2C_DUALADDRESS_DISABLE;
    hi2c3.Init.OwnAddress2      = 0;
    hi2c3.Init.GeneralCallMode  = I2C_GENERALCALL_DISABLE;
    hi2c3.Init.NoStretchMode    = I2C_NOSTRETCH_DISABLE;

    if(HAL_I2C_Init(&hi2c3) != HAL_OK)
    {
      return IOM_ERROR_INTERFACE;
    }

    if (HAL_I2CEx_ConfigAnalogFilter(&hi2c3, I2C_ANALOGFILTER_DISABLE) != HAL_OK) {
      return IOM_ERROR_INTERFACE;
    }

    if (HAL_I2CEx_ConfigDigitalFilter(&hi2c3, 0) != HAL_OK) {
      return IOM_ERROR_INTERFACE;
    }

    if(HAL_I2C_Master_Transmit(&hi2c3, IO_DAC_CH_1_2_ADDR, &IO_DAC_VREF_BUF, 1,
HAL_MAX_DELAY) != HAL_OK) {
      return IOM_ERROR_INTERFACE;
    }

    if(HAL_I2C_Master_Transmit(&hi2c3, IO_DAC_CH_3_4_ADDR, &IO_DAC_VREF_BUF, 1,
HAL_MAX_DELAY) != HAL_OK) {
      return IOM_ERROR_INTERFACE;
    }
```

```
    return IOM_OK;
  }
```

InitTimers() initializes the timers necessary for synchronizing all components of data transmission including DMA and clock lines.

```
IOM_ERROR InitTimers(void) {
    __HAL_RCC_TIM2_CLK_ENABLE();
    __HAL_RCC_TIM5_CLK_ENABLE();
    __HAL_RCC_TIM15_CLK_ENABLE();
    __HAL_RCC_TIM3_CLK_ENABLE();
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_OC_InitTypeDef sConfigOC = {0};
  TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig = {0};
  TIM_ClockConfigTypeDef sClockSourceConfig = {0};

  htim8.Instance = TIM8;
  htim8.Init.Prescaler = 32;
  htim8.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim8.Init.Period = timerPeriod;
  htim8.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim8.Init.RepetitionCounter = 0;
  htim8.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_PWM_Init(&htim8) != HAL_OK)
  {
    return IOM_ERROR_INVALID; //TODO put a better error here
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  HAL_TIMEx_MasterConfigSynchronization(&htim8, &sMasterConfig);


  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 9;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCNPolarity = TIM_OCNPOLARITY_LOW;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
  sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
  if (HAL_TIM_PWM_ConfigChannel(&htim8, &sConfigOC, TIM_CHANNEL_4) != HAL_OK)
  {
    return IOM_ERROR_INVALID; //TODO put a better error here
  }


  sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_DISABLE;
  sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_DISABLE;
  sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_OFF;
  sBreakDeadTimeConfig.DeadTime = 0;
```

```
sBreakDeadTimeConfig.BreakState = TIM_BREAK_DISABLE;
sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_ENABLE;
HAL_TIMEx_ConfigBreakDeadTime(&htim8, &sBreakDeadTimeConfig);


  HAL_NVIC_SetPriority(TIM8_CC_IRQn, 0, 1);
  HAL_NVIC_EnableIRQ(TIM8_CC_IRQn);

  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 32;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = timerPeriod;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    return IOM_ERROR_INTERFACE;
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
  {
    return IOM_ERROR_INTERFACE;
  }
  if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
  {
    return IOM_ERROR_INTERFACE;
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
  {
    return IOM_ERROR_INTERFACE;
  }
  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = timerPulse;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
  {
    return IOM_ERROR_INTERFACE; //TODO put a better error here
  }

  htim5.Instance = TIM5;
  htim5.Init.Prescaler = 32;
  htim5.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim5.Init.Period = timerPeriod;
  htim5.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim5.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim5) != HAL_OK)
  {
    return IOM_ERROR_INTERFACE;
  }
```

```
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim5, &sClockSourceConfig) != HAL_OK)
{
  return IOM_ERROR_INTERFACE;
}
if (HAL_TIM_PWM_Init(&htim5) != HAL_OK)
{
  return IOM_ERROR_INTERFACE;
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim5, &sMasterConfig) != HAL_OK)
{
  return IOM_ERROR_INTERFACE;
}
sConfigOC.OCMode = TIM_OCMODE_PWM1;
sConfigOC.Pulse = timerPulse;
sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
if (HAL_TIM_PWM_ConfigChannel(&htim5, &sConfigOC, TIM_CHANNEL_2) != HAL_OK)
{
  return IOM_ERROR_INTERFACE;
}
htim15.Instance = TIM15;
htim15.Init.Prescaler = 32;
htim15.Init.CounterMode = TIM_COUNTERMODE_UP;
htim15.Init.Period = timerPeriod;
htim15.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim15.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_Base_Init(&htim15) != HAL_OK)
{
  return IOM_ERROR_INTERFACE;
}
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim15, &sClockSourceConfig) != HAL_OK)
{
  return IOM_ERROR_INTERFACE;
}
if (HAL_TIM_PWM_Init(&htim15) != HAL_OK)
{
  return IOM_ERROR_INTERFACE;
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim15, &sMasterConfig) != HAL_OK)
{
  return IOM_ERROR_INTERFACE;
}
sConfigOC.OCMode = TIM_OCMODE_PWM1;
sConfigOC.Pulse = timerPulse;
sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
if (HAL_TIM_PWM_ConfigChannel(&htim15, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
```

```
    {
      return IOM_ERROR_INTERFACE;
    }
    htim3.Instance = TIM3;
    htim3.Init.Prescaler = 32;
    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim3.Init.Period = timerPeriod;
    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
    {
      return IOM_ERROR_INTERFACE;
    }
    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
    {
      return IOM_ERROR_INTERFACE;
    }
    if (HAL_TIM_PWM_Init(&htim3) != HAL_OK)
    {
      return IOM_ERROR_INTERFACE;
    }
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
    {
      return IOM_ERROR_INTERFACE;
    }
    sConfigOC.OCMode = TIM_OCMODE_PWM1;
    sConfigOC.Pulse = timerPulse;
    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
    {
      return IOM_ERROR_INTERFACE;
    }

    return IOM_OK;
}
```

InitDMA() initializes the DMA for scheduling.

```
IOM_ERROR InitDMA(void) {
  __HAL_RCC_DMA2_CLK_ENABLE();

  hdma_dma_generator0.Instance = DMA2_Stream7;
  hdma_dma_generator0.Init.Request = DMA_REQUEST_TIM8_CH4;
  hdma_dma_generator0.Init.Direction = DMA_MEMORY_TO_PERIPH;
  hdma_dma_generator0.Init.PeriphInc = DMA_PINC_DISABLE;
  hdma_dma_generator0.Init.MemInc = DMA_MINC_ENABLE;
  hdma_dma_generator0.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
```

```
  hdma_dma_generator0.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
  hdma_dma_generator0.Init.Mode = DMA_NORMAL;
  hdma_dma_generator0.Init.Priority = DMA_PRIORITY_VERY_HIGH;
  hdma_dma_generator0.Init.FIFOMode = DMA_FIFOMODE_ENABLE;
  hdma_dma_generator0.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_1QUARTERFULL;
  hdma_dma_generator0.Init.MemBurst = DMA_MBURST_SINGLE;
  hdma_dma_generator0.Init.PeriphBurst = DMA_PBURST_SINGLE;
__HAL_LINKDMA(&htim8, hdma[TIM_DMA_ID_CC4], hdma_dma_generator0);
  HAL_DMA_Init(htim8.hdma[TIM_DMA_ID_CC4]);
  HAL_NVIC_SetPriority(DMA2_Stream7_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA2_Stream7_IRQn);
  return IOM_OK;
}
```

InitIOPins() initializes the IO Pins to their default states.

```
IOM_ERROR InitIOPins() {
  if (IOPinsInitialized != 0) {
     return IOM_ERROR_NOT_ALLOWED;
  }
  for (int i = 0; i < NUM_IO_PINS; i++) {
     IO_Pins[i].dataState = IOCFG_DATA_STATE_DISABLED;
     IO_Pins[i].polarity = IOCFG_POLARITY_FALSE;
     IO_Pins[i].idleState = IOCFG_IDLE_STATE_LOW;
     IO_Pins[i].differential = IOCFG_DIFFERENTIAL_DISABLED;
  }
  return IOM_OK;
}
```

InitUART() initialises UART to be used for communication with the GUI. (AD)

```
IOM_ERROR InitUART(void) {
    __HAL_RCC_USART1_CLK_ENABLE();
    HAL_NVIC_SetPriority(USART1_IRQn, 0, 1);
    HAL_NVIC_EnableIRQ(USART1_IRQn);
    hUART.Instance          = USART1;
    hUART.Init.BaudRate     = 115200;
    hUART.Init.WordLength   = UART_WORDLENGTH_8B;
    hUART.Init.StopBits     = UART_STOPBITS_1;
    hUART.Init.Parity       = UART_PARITY_NONE;
    hUART.Init.HwFlowCtl    = UART_HWCONTROL_NONE;
    hUART.Init.Mode         = UART_MODE_TX_RX;
    hUART.Init.OverSampling = UART_OVERSAMPLING_16;
    hUART.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;

    HAL_StatusTypeDef ret = HAL_UART_DeInit(&hUART);
    if (ret != HAL_OK)
    {
        return IOM_ERROR_INVALID; //TODO replace with better error
```

```
    }
    ret = HAL_UART_Init(&hUART);
    if (ret != HAL_OK)
    {
        return IOM_ERROR_INVALID; //TODO replace with better error
    }
  return IOM_OK;
}
```

5.3.5 Scheduler

In order to ensure the input and output buffers do not run out of space on the I/O Master, a scheduler block is necessary to monitor them. This scheduler will run as the main loop of the I/O Master microcontroller after it has been initialized. Figure 60 shows a flowchart of the scheduler main loop.

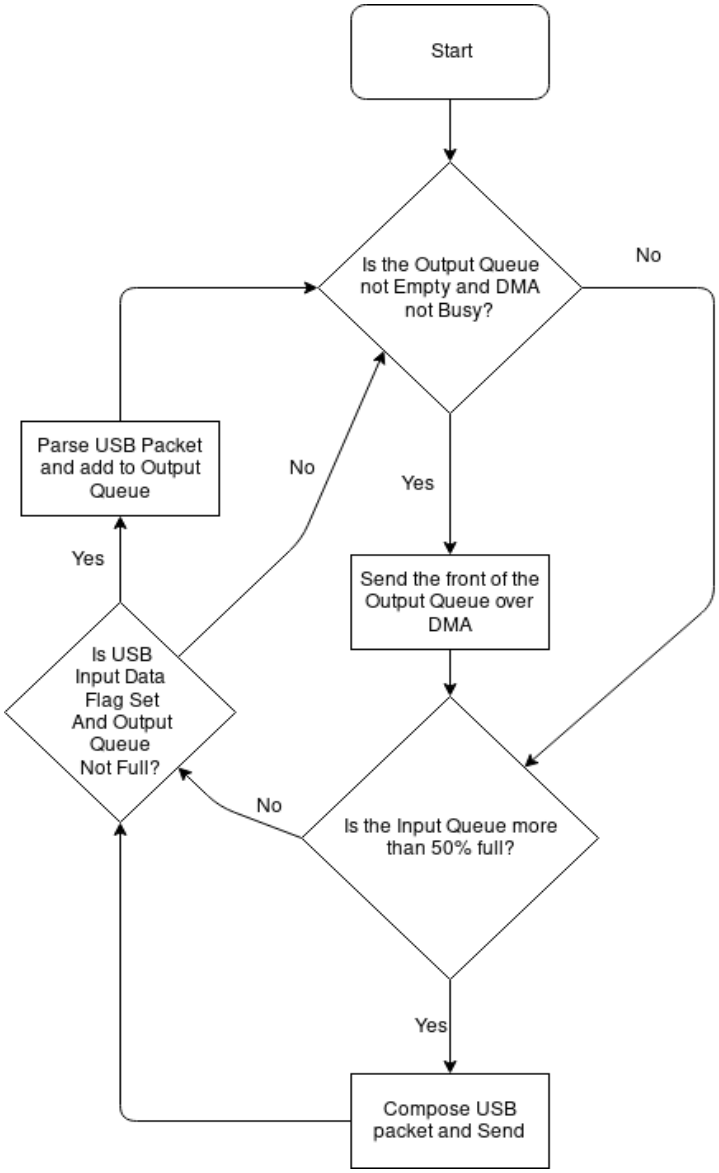


Figure 60: Scheduler Block Main Loop Flowchart

There are four things the scheduler main loop checks. The first is if the *Output Queue* is empty or not. If it is not empty, it will start another DMA stream if DMA is available. Then, if the *Input Queue* is more than 50% full, the data will be composed into a USB packet and sent to the

connected computer. Finally, if there is data that was read in over USB waiting to be parsed and there is space in the *Output Queue*, the data will get parsed and queued.

These four checks are designed to be quick to test and still allow continuous functionality. If the *Output Queue* fills up and cannot keep up with sending data, then the I/O Master will stop processing USB packets. If the *Output Queue* is not full, then USB packets can be processed and data can be added to the queue. When the *Input Queue* is half full, the data will be dumped into a USB packet, thus avoiding the *Input Queue* from completely filling up.

In addition to the main scheduler loop, there are some interrupts that will pause the loop. If an interrupt is called, the CPU will set a flag and return to the main loop. The interrupts include a DMA transfer complete interrupt that turns off the DMA busy flag, an interrupt for an incoming USB data packet that sets a data available flag, and an interrupt tied to a timer to shift data out of the DMA stream and into the *Input Queue* for reading data. (CS)

The current implementation of the main loop scheduler splits the actions based on the current state in order to limit the number of decisions that are needed to be made. The implementation is fully functional for sending data. This loop does not contain much code because it must quickly decide what to do without wasting system resources.

```
switch(IOMState) {
```

If the MCU is in the Init state and in the main loop, then it is done initializing and can be moved to the configuration state.

```
    case IOM_STATE_INIT:
    IOMState = IOM_STATE_CONF;
```

```
break;
```

If the MCU is in the Busy state, the DMA is checked to see if it is busy. If DMA is not busy and there is data to send, then the data will be sent out. Since this is the first thing that is checked, sending data is the highest priority.

```
case IOM_STATE_BUSY:
if (DMABusyFlag == 0) {
	if (output_buf_queue_size > 0) {
	DMABusyFlag = 1;
	SendOutputData();
	}
```

If DMA is not busy and there isn't any more data to send, then the I/O Master will go back to the Ready state.

```
	else {
	IOMState = IOM_STATE_READY:
	}
}
```

Afterwards, if the I/O Master is in Busy, Configuration, or Ready state and there is nothing to be sent, a received command will be processed if there are any. Otherwise, nothing happens.

```
case IOM_STATE_CONF:
case IOM_STATE_READY:
if (command_buf_queue_size > 0) {
ProcessCommand();
}
break;
```

Currently, if the I/O Master is put into an Error state, the Red Status LED will blink indefinitely. This can be expanded in the future to allow for more precise error logging and error recovery. (CS)

```
case IOM_STATE_ERROR:
	HAL_GPIO_TogglePin(STATUS_R_GPIO_Port, STATUS_R_GPIO_Pin);
```

```
      break;
}
```

5.3.6 Data Route Handler

The data route handler component includes the parts of the microcontroller firmware that stream data in and out. To input and output data over GPIO, the DMA stream must be configured to the corresponding data. Reading and writing data use the same process, but with the source and destination switched. The data route handler will be informed from the scheduler as to what data to send and receive, and the corresponding hardware interface will be called. In the case of USB, the USB Data Buffer component can be called directly through the data route handler. For DMA, some additional steps are required to prepare the DMA for streaming. The main steps are already done during initialization, but the data source must still be set up before running.

The *StartDMATransfer* function is implemented to send data. When *SendOutputData* is called, the front of the queue is brought into *StartDMATransfer*.

```
IOM_ERROR SendOutputData(void) {
  if (output_buf_queue_size == 0) {
     return IOM_ERROR_QUEUE_EMPTY;
  }

  IOM_Output_Buffer* bufferData = &(output_buf_queue[output_buf_queue_out_ptr]);
  output_buf_queue_out_ptr = (output_buf_queue_out_ptr + 1) %
OUTPUT_BUF_QUEUE_MAX_SIZE;
  output_buf_queue_size--;

  StartDMATransfer(bufferData);
  return IOM_OK;
}
```

Note that bufferData is a pointer. However, this pointer is not being allocated since it is a circular buffer. As a result, There is no need to free this memory.

StartDMATransfer first configures the DMA for the correct data source and output. Since the DMA clock is not running yet, the HAL library function *HAL_DMA_Start* is utilized. Then, the DMA timer channel is enabled with *__HAL_TIM_ENABLE_DMA*. At this point, the DMA is technically started, but since the connected timer is not running yet, nothing will happen.

```
void StartDMATransfer(IOM_Output_Buffer* pBuffer) {
  bytesToSend = pBuffer->length;
  HAL_DMA_Start(htim8.hdma[TIM_DMA_ID_CC4], (uint32_t)pBuffer->data,
(uint32_t)(&(GPIOD->ODR)), pBuffer->length);
  __HAL_TIM_ENABLE_DMA(&htim8, TIM_DMA_CC4);
```

Due to needing to start both the clock timers and the DMA output stream at as close to the same time as possible, the HAL library cannot be used. In testing, the HAL library was used and it was found to have too much delay between the clock signal and data signal. Rather, the timer registers are edited directly. *TIM8* is the timer connected to control DMA. These lines of code clear the enabled registers for the timer, then enable channel 4 for the timer and the corresponding interrupt. Since the DMA interrupt does not work, the timer interrupt can be used to determine when all of the data has been sent.

```
TIM8->CCER &= ~(TIM_CCER_CC4E);
TIM8->CCER |= TIM_CCER_CC4E;
TIM8->BDTR |= TIM_BDTR_MOE;
TIM8->DIER |= TIM_DIER_CC4IE;
```

Since each pin on the I/O Master can be dynamically configured as an input, output, or clock, each pin must be checked to see what it is set as and configure it accordingly before sending data. First, all of the clock pins have their timers setup.

```
TIM_TypeDef* pTim;
for (int i = 0; i < 4; i++) {
   if (IO_Pins[i].dataState == IOCFG_DATA_STATE_CLOCK) {
   GetIOPinTimer(i + 1, pTim);
   if (pTIM == TIM5) {
```

```
            pTim->CCER &= ~(TIM_CCER_CC2E);
      } else {
            pTim->CCER &= ~(TIM_CCER_CC1E);
      }

      pTim->BDTR |= TIM_BDTR_MOE;
      pTim->CNT = 5;
      }
```

Then, the chip select pins are set to their active state (opposite of their idle state). Afterwards, all other pins are set to their idle state.

```
      else if (IO_Pins[i].dataState == IOCFG_DATA_STATE_CS) {
      if (IO_Pins[i].idleState == IOCFG_IDLE_STATE_HIGH) {
      //Drive the pin low
      IOPIN_GPIO_OUTPUT_PORT->ODR &= ~(GetIOPinOutputMask(i + 1));
      } else {
      //Drive the pin high
      IOPIN_GPIO_OUTPUT_PORT->ODR  |= (GetIOPinOutputMask(i + 1));
      }
      } else {
      if (IO_Pins[i].idleState == IOCFG_IDLE_STATE_HIGH) {
      //Set default to high
      IOPIN_GPIO_OUTPUT_PORT->ODR |= (GetIOPinOutputMask(i + 1));
      }  else {
      //Set default to low
      IOPIN_GPIO_OUTPUT_PORT->ODR &= ~(GetIOPinOutputMask(i + 1));
      }
      }

   }
```

The last step is to set the count of the DMA timer to immediately cause a rising edge and enable its timer.

```
   TIM8->CNT = 5;
   TIM8->CR1 |= TIM_CR1_CEN;
   }
```

The clock timers are not enabled because the data output must be changed before the clock changes. To achieve this, the other clocks are enabled in the *TIM8* interrupt.

```
 void TIM8_CC_IRQHandler() {
```

```
   //Enable clocks
   TIM_TypeDef* pTim;
   for (int i = 0; i < 4; i++) {
      if (IO_Pins[i].dataSate == IOCFG_DATA_STATE_CLOCK) {
      GetIOPinTimer(i + 1, pTim);
      //TIM5 uses channel 2, all others use channel 1
      if (pTim == TIM5) {
      pTim->CCER |= TIM_CCER_CC2E;
      } else {
      pTim->CCER |= TIM_CCER_CC1E;
      }
```

Once the DMA stream has finished sending its last byte, the clock timers are all disabled.

```
      if (bytesToSend == 1) {
      //Disable timer if there is no more data to send
      pTim->CR1 &= ~(TIM_CR1_CEN);
      pTim->CNT = 5;
      }
      }
   }
```

The *TIM8* interrupt flag must be disabled so the interrupt can be triggered again. If the last byte has been sent, then *TIM8* is disabled, the DMA is reset, and the busy flag is turned off. Otherwise, the *bytesToSend* variable decrements and the interrupt is exited. (CS)

```
   //Disable flag
   TIM8->SR &= ~(TIM_SR_CC4IF);
   if (bytesToSend == 1) {
      TIM8->CR1 &= ~(TIM_CR1_CEN);
      ResetDMA();
      DMABusyFlag = 0;
   } else {
   bytesToSend--;
   }
}
```

The data route code for streaming data over DMA was tested to ensure that data could be written to every pin, a variable amount of data can be written, and that the chip select is activated

correctly. Testing was accomplished through communications with an SPI LCD screen, along with oscilloscope waveform and voltage measurements.

5.3.7 Data Parser / Composer

To be able to communicate with other devices, the Data Parser and Data Composer components are responsible for making sense of data sent to the I/O Master and formatting data to send off. This process needs to be relatively high speed so that data can continue to be streamed through the I/O Master. To accomplish such a task, a command structure has been designed and implemented in both the firmware and the GUI. The commands are further explained in Section 5.3.8.

Data that is communicated between the GUI and the firmware consists of board configurator data and I/O data. Board configuration does not need to operate at high speed, so the implementation focuses on ease of use and clarity over optimizing the speed of processing configuration commands.

In the implementation, commands are read in through UART and processed into command structures. UART is currently used for the design in order to get the I/O Master fully functional with minimal work time. Since implementing high speed USB is a lengthy process, it was decided to tackle such a task after finishing the other main design requirements of the I/O Master.

Commands are read into a command buffer, which is implemented as a circular queue buffer. This can store a total of 512 bytes. The queue size and data are set as volatile variables since they

can be updated from an interrupt. This ensures the compiler does not make any assumptions about the value of these variables and explicitly reads them every time they are used. (CS)

```
#define COMMAND_BUF_QUEUE_MAX_SIZE 512
volatile uint8_t command_buf_queue[COMMAND_BUF_QUEUE_MAX_SIZE];
uint16_t command_buf_in_ptr;
uint16_t command_buf_out_ptr;
volatile uint16_t command_buf_queue_size;
IOM_Output_Buffer current_command = {};
```

UART data is automatically fed into the command buffer queue through an interrupt that triggers the *QueueCommand* function.

```
IOM_ERROR QueueCommand(uint8_t* pCommand) {
     if (command_buf_queue_size >= COMMAND_BUF_QUEUE_MAX_SIZE) {
     return IOM_ERROR_QUEUE_FULL;
     }
     command_buf_queue[command_buf_in_ptr] = *pCommand;
     free(pCommand);
     command_buf_in_ptr = (command_buf_in_ptr + 1) % COMMAND_BUF_QUEUE_MAX_SIZE;
     command_buf_queue_size++;
     return IOM_OK;
}
```

Commands are processed when the scheduler calls the *ProcessCommand* function. This function will walk through the command buffer and build the *current_command* variable.

```
IOM_ERROR ProcessCommand() {
     if (command_buf_queue_size == 0) {
     return IOM_ERROR_QUEUE_EMPTY;
     }
     while (command_buf_queue_size > 0) {
     //All commands are at least 2 bytes in length
     //If there are not 2 bytes yet, just load the data
     switch(current_command.length) {
            case 0:
                  current_command.data = &command_buf_queue[command_buf_out_ptr];
                  //no break, fall through to 1
            case 1:
                  current_command.length++;
                  continue;
            default:
                  break;
```

```
    }
```

Then, the command is categorized based on the command type. All of the existing command types are part of an enumeration called *IOM_CMD_TYPE*. Commands are further explained in the Board Configurator section.

```
typedef enum {
    IOM_CMD_TYPE_STATE = 0x01,
    IOM_CMD_TYPE_CONFIGURATION = 0x02,
    IOM_CMD_TYPE_DATA = 0x03,
} IOM_CMD_TYPE;
```

After the command is categorized by the type, the second byte is read to figure out the exact command. Each command type has its own enumeration for each command that exists.

```
typedef enum {
    IOM_STATE_CMD_GET = 0x00,
    IOM_STATE_CMD_SET_CONFIG = 0x01,
    IOM_STATE_CMD_SET_DATA = 0x02,
    IOM_STATE_CMD_SET_RUN = 0x03,
    IOM_STATE_CMD_SET_STOP = 0x04,
} IOM_STATE_CMD;

typedef enum {
    IOM_CONFIG_CMD_GET_TARGET_DEVICE_VOLTAGE = 0x00,
    IOM_CONFIG_CMD_SET_TARGET_DEVICE_VOLTAGE = 0x01,
    IOM_CONFIG_CMD_GET_PIN_PARAMETERS = 0x02,
    IOM_CONFIG_CMD_SET_PIN_PARAMETERS = 0x03,
    IOM_CONFIG_CMD_GET_DATA_SPEED = 0x04,
    IOM_CONFIG_CMD_SET_DATA_SPEED = 0x05,
    IOM_CONFIG_CMD_GET_DIFFERENTIAL_MODE = 0x06,
    IOM_CONFIG_CMD_SET_DIFFERENTIAL_MODE = 0x07,
    IOM_CONFIG_CMD_GET_PIN_MODE = 0x08,
    IOM_CONFIG_CMD_SET_PIN_MODE = 0x09,
    IOM_CONFIG_CMD_GET_DATA_PARAMETERS = 0x0A,
    IOM_CONFIG_CMD_SET_DATA_PARAMETERS = 0x0B,
    IOM_CONFIG_CMD_GET_PIN_LOGIC_LEVELS = 0x0C,
    IOM_CONFIG_CMD_SET_PIN_LOGIC_LEVELS = 0x0D,
} IOM_CONFIG_CMD;

typedef enum {
    IOM_DATA_CMD_SET_DATA = 0x00,
    IOM_DATA_CMD_SET_RECEIVE_BYTES = 0x01,
```

```
} IOM_DATA_CMD;
```

Next, the command processor will check if all of the command's data has been received yet. If not, then it will exit and try to parse again once there is more data. All of the command sizes are listed as preprocessor directives.

```
//Command sizes (in bytes)
#define IOM_STATE_CMD_GET_SIZE 2
#define IOM_STATE_CMD_SET_CONFIG_SIZE 2
#define IOM_STATE_CMD_SET_DATA_SIZE 2
#define IOM_STATE_CMD_SET_RUN_SIZE 2
#define IOM_STATE_CMD_SET_STOP_SIZE 2

#define IOM_CONFIG_CMD_GET_TARGET_DEVICE_VOLTAGE_SIZE 2
#define IOM_CONFIG_CMD_SET_TARGET_DEVICE_VOLTAGE_SIZE 4
#define IOM_CONFIG_CMD_GET_PIN_PARAMETERS_SIZE 3
#define IOM_CONFIG_CMD_SET_PIN_PARAMETERS_SIZE 4
#define IOM_CONFIG_CMD_GET_DATA_SPEED_SIZE 2
#define IOM_CONFIG_CMD_SET_DATA_SPEED_SIZE 4
#define IOM_CONFIG_CMD_GET_DIFFERENTIAL_MODE_SIZE 3
#define IOM_CONFIG_CMD_SET_DIFFERENTIAL_MODE_SIZE 4
#define IOM_CONFIG_CMD_GET_PIN_MODE_SIZE 3
#define IOM_CONFIG_CMD_SET_PIN_MODE_SIZE 4
#define IOM_CONFIG_CMD_GET_DATA_PARAMETERS_SIZE 3
#define IOM_CONFIG_CMD_SET_DATA_PARAMETERS_SIZE 4
#define IOM_CONFIG_CMD_GET_PIN_LOGIC_LEVELS_SIZE 3
#define IOM_CONFIG_CMD_SET_PIN_LOGIC_LEVELS_SIZE 7

#define IOM_DATA_CMD_SET_DATA_SIZE 4
#define IOM_DATA_CMD_SET_RECEIVE_BYTES_SIZE 3
```

Each command is tested, and if it passes the identifier and command length tests, then the command is executed. The pattern for this process is nearly identical for every command, making it easy to read and add additional commands in the future. If a command fails, the process will run again once there is more data to process. (CS)

```
switch(*(current_command.data)) {
	case IOM_CMD_TYPE_STATE:
		switch(*(current_command.data + 1)) {
		case IOM_STATE_CMD_GET:
			if (current_command.length == IOM_CONFIG_CMD_GET_SIZE) {
			GetState();
```

```
                        }
                        break;
                case IOM_STATE_CMD_SET_CONFIG:
                        if (current_command.length ==
IOM_CONFIG_CMD_SET_CONFIG_SIZE) {
                        SetConfigState();
                        }
                        break;
                case IOM_STATE_CMD_SET_RUN:
                        if (current_command.length == IOM_CONFIG_CMD_SET_RUN_SIZE)
{
                        SetConfigState();
                        }
                        break;
                case IOM_STATE_CMD_SET_STOP:
                        if (current_command.length ==
IOM_CONFIG_CMD_SET_STOP_SIZE) {
                        SetConfigState();
                        }
                        break;
                }
                break;
            case IOM_CMD_TYPE_CONFIGURATION:
                switch(*(current_command.data + 1)) {
                case IOM_CONFIG_CMD_GET_TARGET_DEVICE_VOLTAGE:
                        if (current_command.length ==
IOM_CONFIG_CMD_GET_TARGET_DEVICE_VOLTAGE_SIZE) {
                        GetTargetDeviceVoltage();
                        }
                        break;
                case IOM_CONFIG_CMD_SET_TARGET_DEVICE_VOLTAGE:
                        if (current_command.length ==
IOM_CONFIG_CMD_GET_TARGET_DEVICE_VOLTAGE_SIZE) {
                        SetTargetDeviceVoltage();
                        }
                        break;
                case IOM_CONFIG_CMD_GET_PIN_PARAMETERS:
                        if (current_command.length ==
IOM_CONFIG_CMD_GET_PIN_PARAMETERS_SIZE) {
                        GetPinParameters();
                        }
                        break;
                case IOM_CONFIG_CMD_SET_PIN_PARAMETERS:
                        if (current_command.length ==
IOM_CONFIG_CMD_SET_PIN_PARAMETERS_SIZE) {
                        SetPinParameters();
                        }
                        break;
                case IOM_CONFIG_CMD_GET_DATA_SPEED:
                        if (current_command.length ==
IOM_CONFIG_CMD_GET_DATA_SPEED_SIZE) {
                        GetDataSpeed();
                        }
```

```
                        break;
                case IOM_CONFIG_CMD_SET_DATA_SPEED:
                        if (current_command.length ==
IOM_CONFIG_CMD_SET_DATA_SPEED_SIZE) {
                        SetDataSpeed();
                        }
                        break;
                case IOM_CONFIG_CMD_GET_DIFFERENTIAL_MODE:
                        if (current_command.length ==
IOM_CONFIG_CMD_GET_DIFFERENTIAL_MODE_SIZE) {
                        GetDifferentialMode();
                        }
                        break;
                case IOM_CONFIG_CMD_SET_DIFFERENTIAL_MODE:
                        if (current_command.length ==
IOM_CONFIG_CMD_SET_DIFFERENTIAL_MODE_SIZE) {
                        SetDifferentialMode();
                        }
                        break;
                case IOM_CONFIG_CMD_GET_PIN_MODE:
                        if (current_command.length ==
IOM_CONFIG_CMD_GET_PIN_MODE_SIZE) {
                        GetPinMode();
                        }
                        break;
                case IOM_CONFIG_CMD_SET_PIN_MODE:
                        if (current_command.length ==
IOM_CONFIG_CMD_SET_PIN_MODE_SIZE) {
                        SetPinMode();
                        }
                        break;
                case IOM_CONFIG_CMD_GET_DATA_PARAMETERS:
                        if (current_command.length ==
IOM_CONFIG_CMD_GET_DATA_PARAMETERS_SIZE) {
                        GetDataParameters();
                        }
                        break;
                case IOM_CONFIG_CMD_SET_DATA_PARAMETERS:
                        break;
                        if (current_command.length ==
IOM_CONFIG_CMD_SET_DATA_PARAMETERS_SIZE) {
                        SetDataParameters();
                        }
                case IOM_CONFIG_CMD_GET_PIN_LOGIC_LEVELS:
                        break;
                        if (current_command.length ==
IOM_CONFIG_CMD_GET_PIN_LOGIC_LEVELS_SIZE) {
                        GetPinLogicLevels();
                        }
                case IOM_CONFIG_CMD_SET_PIN_LOGIC_LEVELS:
                        if (current_command.length ==
IOM_CONFIG_CMD_SET_PIN_LOGIC_LEVELS_SIZE) {
                        SetPinLogicLevels();
```

```
                        }
                        break;
                  }
                  break;
            case IOM_CMD_TYPE_DATA:
                  switch(*(current_command.data + 1)) {
                  case IOM_DATA_CMD_SET_DATA:
                        if (current_command.length >= IOM_DATA_CMD_SET_DATA_SIZE)
{
                        SetData();
                        }
                  break;
                  case IOM_DATA_CMD_SET_RECEIVE_BYTES:
                        if (current_command.length ==
IOM_DATA_CMD_SET_RECEIVE_BYTES_SIZE) {
                        SetReceiveBytes();
                        }
                  break;
                  }
                  break;
     }
     }

}
```

In order for DMA to move data with GPIO, the data must be placed in a memory location that DMA can access and the data must be formatted to match the block size that DMA streams. The I/O Master will use memory domain D2, which DMA has access to read and write. Memory domain D2 starts at address *0x30000000*. The process of parsing and composing data is similar, but reversed. The implementation of composing data for writing is explained in detail, and the process can be reversed to implement parsing data for reading.

The following preprocessor directives are used in the code to configure a size and memory address for a DMA output buffer. This buffer is for writing data out the GPIO pins.

```
#define DMA_OUTPUT_BUF_MAX_SIZE 1024 ///< A max size so the buffer memory domain
doesn't fill up
#define DMA_WRITE_BUF_MIN_ADDR 0x30000000 ///< The starting address of the DMA
write buffer
#define DMA_WRITE_BUF_MAX_ADDR (RAM_D2_MIN_ADDR + DMA_OUTPUT_BUF_MAX_SIZE) ///< The
ending address of the DMA Write buffer
```

```
#define OUTPUT_BUF_QUEUE_MAX_SIZE 10 ///< The size of the DMA output buffer queue
```

The default linker script to compile and flash the microcontroller does not have any way to allocate data on the heap or stack for memory domain D2. The only way to write data to memory domain D2 is to directly point a pointer to it and track it manually. Other methods would require modifying the linker script.

The DMA output buffer has two pointers used for tracking data output. The *in* pointer tracks the address where new data is to be added to the buffer, and the *out* pointer tracks the current point of data to be streamed from the buffer to GPIO over DMA. These two pointers are defined

```
uint8_t* output_buf_in_ptr = DMA_WRITE_BUF_MIN_ADDR; ///< Pointer to add data to
the buffer
uint8_t* output_buf_out_ptr = DMA_WRITE_BUF_MIN_ADDR; ///< Pointer to remove data
from the buffer
```

Since there is a need to control where the data is stored to restrict it to the range within the defined output buffer, the standard dynamic memory allocation function *malloc*, which is used to allocate memory on the heap, cannot be used. The *malloc* function offers no flexibility to choose from a subset of heap memory. Therefore, a custom memory allocation process is necessary for preparing the output buffer. The custom memory allocation process is handled when memory is being queued into the buffer.

A struct named *IOM_Output_Buffer* is defined to retain information related to the location of the block of data to send and the size of the block.

```
struct IOM_Output_Buffer {
  uint8_t* data; ///< The location of data in the buffer
  size_t length; ///< The length of the data
}
```

A queue of *IOM_Output_Buffer* objects keeps track of what block of data to send over DMA next. The I/O Master implements this queue using an array, tracking the index for adding to the queue, the index for removing from the queue, and the size of the queue.

```
IOM_Output_Buffer output_buf_queue[OUTPUT_BUF_QUEUE_MAX_SIZE];
uint8_t output_buf_queue_in_ptr = 0;
uint8_t output_buf_queue_out_ptr = 0;
uint8_t output_buf_queue_size = 0;
```

A function named *QueueOutputDataToSend* writes a block of data to the output buffer and queues it to be sent out. The pin number that the data will be sent out is specified with *pinNum*.

```
///
/// @brief puts a block of data into the output buffer
///
/// @param pData, pointer to the first byte of the data set
///
/// @param length, the length of the data to set, in bytes
///
/// @param pinNum, which pin to set
///
/// @return IOM_OK if OK, IOM_ERROR_BUFFER_FULL if
/// there is no room left in the output buffer, IOM_ERROR_QUEUE_FULL
/// if no room left in the output queue
///
IOM_ERROR QueueOutputDataToSend(uint8_t* pData, size_t length, uint8_t pinNum);
```

The first thing the function does is check if there is enough space in the output buffer to allocate the amount of data to be queued. The data block cannot be split up, as it would cause a delay in transmission. If there is not enough space available, the *IOM_ERROR_BUFFER_FULL* error is returned.

```
   /// Check if the data to send can't fit in the end of the buffer
   if ((output_buf_in_ptr + (length * 8)) > RAM_D2_MAX_ADDR){
      /// If it can't, see if it can fit in the beginning
      if ((output_buf_out_ptr - RAM_D2_MIN_ADDR) >= (length * 8)) {
      output_buf_in_ptr = RAM_D2_MIN_ADDR;
      } else {
      return IOM_ERROR_BUFFER_FULL;
      }
```

```
}
```

Next, the function ensures that the output buffer queue has a slot open to queue in the data. If there is already the maximum number of blocks queued, the *IOM_ERROR_QUEUE_FULL* error is returned.

```
//Make sure the output buffer queue has space available
if (output_buf_queue_size >= OUTPUT_BUF_QUEUE_MAX_SIZE) {
   return IOM_ERROR_QUEUE_FULL;
}
```

Now that it has been established that there is sufficient space to queue the block of data, the data is formatted to be outputted over DMA. This process splits every bit of data into its own byte in the output buffer and writes it to the output buffer. It is necessary to split the data this way as the smallest amount of data that DMA can stream is one byte. The pinNum input parameter determines where to put the data into the output buffer.

The *output_buf_queue* has the data queued in, then the process for formatting and loading the data into the buffer begins.

```
//Add the data to be output to the queue
output_buf_queue[output_buf_queue_in_ptr].data = output_buf_in_ptr;
output_buf_queue[output_buf_queue_in_ptr].length = length * 8;

*(output_buf_in_ptr) = idle_pin_output;
//Iterate through each byte
for (size_t i = 0; i < length; i++) {
```

The first thing to check is if there is a data polarity for the specified pin. If there is, then all of the data needs bit-flipped.

```
//Check if the bits need flipped
if (IO_Pins[pinNum-1].polarity == IOCFG_POLARITY_TRUE) {
  *(pData + i) = ~*(pData + 1);
}
```

```
    //Iterate through each bit
```

Then, each bit is compared to the idle state. If the Idle state is high and the bit is a *1*, then the output buffer is assigned a *0*. Otherwise, the output buffer is assigned a *1*. The opposite happens if the idle state for the pin is low.

```
   for (size_t j = 0; j < 8; j++) {
      //Set the byte in the output buffer to the current bit of data
      if (IO_Pins[pinNum-1].idleState == IOCFG_IDLE_STATE_HIGH) {
      //Idle high, set bits to low
      *(output_buf_in_ptr) &= (((*(pData+i) >> (7 - j))& 0x01) == 0) ?
~((GetIOPinOutputMask(pinNum) + 1)) : 0xFF;
      } else {
      //Idle low, set bits to high
      *(output_buf_in_ptr) |= ((*(pData+i) >> (7 - j)) <<
(GetIOPinOutputPos(pinNum))) & GetIOPinOutputMask(pinNum);
      }
```

The final step is to encode in any active chip select data. Since data is being sent, the chip select should be enabled. chip select pins will be set to the opposite of their idle state.

```
      //Write in Chip Select Data
      for (int i = 0; i < 4; i++) {
      if (IO_Pins[i].dataState == IOCFG_DATA_STATE_CS) {
      if (IO_Pins[i].idleState == IOCFG_IDLE_STATE_HIGH) {
            *(output_buf_in_ptr) &= ~((GetIOPinOUtputMask(i+1) + 1));
      } else {
            //Active low
            *(output_buf_in_ptr) |= (1 << GetIOPinOutputPos(i + 1)) &
GetIOPinOutputMask(i + 1);
      }
      }
      }
```

The output buffer data pointer is incremented to the next byte and the process repeats until all of the data has been put into the output buffer.

```
      output_buf_in_ptr++;
      *(output_buf_in_ptr) = idle_pin_output;
   }
```

```
  }
```

After the data has been written to the output buffer, the input pointer and the size of the buffer are incremented. Since the operation was successful, the function returns *IOM_OK*.

```
 output_buf_queue_in_ptr = (output_buf_queue_in_ptr + 1) %
OUTPUT_BUF_QUEUE_MAX_SIZE;
 output_buf_queue_size ++;
 return IOM_OK;
}
```

The smallest chunk of data the DMA controller can be configured to send is one byte. Since the I/O Master supports a maximum of four data outputs, only a maximum of four bits will have usable data for every byte sent. If less than four data output lines are desired, then the unused GPIO pins can be turned off so data is not outputted.

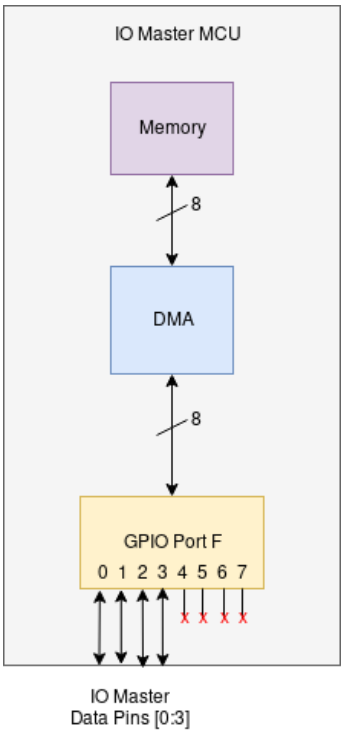


Figure 61: Diagram of Data Flow from Memory to GPIO via DMA

As shown in Figure 61, GPIO 4:7 are disconnected. Because DMA must send at least 1 byte to GPIO F, data is written to GPIO 4:7, but the pins are unused (identified with a red X in the figure). Only GPIO 0:3 will be connected to I/O Master data pins. If only a subset of GPIO 0:3 are desired to be used, then the unused pins can be configured to be disabled for writing.

There are potential cons to consider due to the restriction of sending a whole byte over DMA. One downside is that four of the eight GPIO pins must be unused. Since there are over 100 GPIO pins on the microcontroller, blocking four GPIO pins is insignificant and not a concern.

There is also a space complexity cost for storing and transmitting data. The worst case scenario is that only one of the four I/O data lines are being used to send data. In this scenario, for every one bit of data to send or receive, eight bits of storage are necessary to properly format and represent the data on the microcontroller. Thus, the space complexity is O(8n) in algorithmic Big O notation, which is linear. O(8n) is effectively the same as O(n), so the space complexity is not a concern. In terms of data transmission, there is no additional time cost added by needing to send a whole byte because the DMA sends the entire byte in parallel through the bus. Sending a single bit (which is not actually possible) or a whole byte is the same time cost. The time complexity could be seen for sending data along the bus as a constant time, or O(1).

The benefits of DMA are significant. Data can be streamed without needing to use CPU clock cycles. While there is some set up time to prepare the data before it can be sent over DMA and vice versa, the CPU can easily keep up with this process as the time complexity of formatting the data for writing or processing the data for reading is linear, on the order of a constant times the number of bits to send. Given that the selected microcontroller has a CPU clock that is more than

10 times the maximum transmission speed of 10 MHz, the microcontroller will not be a bottleneck. (CS)

5.3.8 Board Configurator

The board configurator is in charge of controlling all of the hardware settings on the I/O Master. This includes setting the level shifter levels, frequencies, pin modes, pin idle states, polarity, pull-up and pull-down resistors, differential mode, and the target device voltage. In order to communicate these settings to the I/O Master, a standardized protocol of commands is necessary. Tables 50, 51, and 53 contain the three types of commands that exist. Each command has a two byte identifier. The first byte tells the command's type, and the second byte is the ID for that command type. While this format is not optimized to minimize bytes transferred, it allows for flexibility for the addition of many new commands in the future while maintaining backwards compatibility of the currently defined commands. (CS)

Table 50: State Commands

Byte 1	**Byte 2**	**Command Name**
0x01	0x00	Get State
0x01	0x01	Set Config State
0x01	0x02	Set Data State
0x01	0x03	Set Run State
0x01	0x04	Set Stop State

Table 50 lists the state commands for the I/O Master. These commands are used to control what state the I/O Master is in, along with checking to see what state it is currently in. *Set Data State* is analogous to setting the I/O Master to the Ready state in the state machine. *Set Run State* will

put the I/O Master into the Busy state if there is data to send. *Set Stop State* will stop the I/O master if it is currently sending or receiving data, putting it back into the Ready state. The current implementation limits what state changes are valid, and additional rules can be enforced in future implementations to make the design safer. (CS)

```
void ProcessStateCommand(IOM_Output_Buffer buffer) {
     current_command[1] = *(buffer.data + 1);
     free(buffer.data);
     switch(current_command[1]) {
     case 1:
            if (IOMState == IOM_STATE_READY) {
                  IOMState = IOM_STATE_CONF;
            }
            break;
     case 2:
            if (IOMState == IOM_STATE_CONF) {
                  IOMState = IOM_STATE_READY;
            }
            break;
     case 3:
            if (IOMState == IOM_STATE_READY) {
                  IOMState = IOM_STATE_BUSY;
            }
            break;
     case 4:
            if (IOMState == IOM_STATE_BUSY)
            IOMState = IOM_STATE_READY;
            break;
     default:
            break;
     }
     command_status = IOM_CS_NEW;
     uint8_t* pData = malloc(2);
     UARTQueueRXData(pData, 2);
     return;
}
```

Table 51: Configuration Commands

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Command Name
0x02	0x00	N/A	N/A	N/A	N/A	N/A	Get Target Device Voltage
0x02	0x01	V_U	V_L	N/A	N/A	N/A	Set Target Device Voltage
0x02	0x02	pinNum	N/A	N/A	N/A	N/A	Get Pin Parameters
0x02	0x03	pinNum	resistorStates	N/A	N/A	N/A	Set Pin Parameters
0x02	0x04	N/A	N/A	N/A	N/A	N/A	Get Data Speed
0x02	0x05	Speed_U	Speed_L	N/A	N/A	N/A	Set Data Speed
0x02	0x06	pinNum	N/A	N/A	N/A	N/A	Get Differential Mode
0x02	0x07	pinNum	on/off	N/A	N/A	N/A	Set Differential Mode
0x02	0x08	pinNum	N/A	N/A	N/A	N/A	Get Pin Mode
0x02	0x09	pinNum	mode	N/A	N/A	N/A	Set Pin Mode
0x02	0x0A	pinNum	N/A	N/A	N/A	N/A	Get Data Parameters
0x02	0x0B	pinNum	idleState & Polarity	N/A	N/A	N/A	Set Data Parameters
0x02	0x0C	pinNum	N/A	N/A	N/A	N/A	Get Pin Logic Levels
0x02	0x0D	pinNum	VH_U	VH_L	VL_U	VL_L	Set Pin Logic Levels
0x02	0x0E	N/A	N/A	N/A	N/A	N/A	Reserved
0x02	0x0F	N/A	N/A	N/A	N/A	N/A	Reserved

Table 51 lists the configuration commands for the IO Master. These commands are only to be sent while the IO Master is in the Config state. Notice each configuration setting has a get and set command. Any parameter that is followed by a _U or _L represents the upper or lower byte of a single value being sent to the IO Master. *Set Target Device Voltage* gives a 2 byte value (V_U and V_L) that represents the voltage set by the MCU DAC. *Set Pin Parameters* gives the Pin number (pinNum) of the pin to be reconfigured and a 2 bit pattern for the desired resistor states. These bit patterns are shown in Table 52.

Table 52: Resistor States for Set Pin Parameters

Bit Pattern	Pull Up Resistor	Pull Down Resistor	Tristate
0b00	Disabled	Disabled	Disabled
0b01	Enabled	Disabled	Disabled
0b10	Disabled	Enabled	Disabled
0b11	Disabled	Disabled	Enabled

Set Pin Mode gives the Pin number of the pin to be reconfigured and a 3 bit pattern for the desired mode. These are seen in the enumerator IOCFG_DATA_STATE. *Set Pin Mode* gives the Pin number of the pin to be reconfigured and a 3 bit pattern for the desired idle state & polarity. These are seen in the enumerators IOCFG_IDLE_STATE and IOCFG_POLARITY respectively. *Set Pin Logic Levels* gives two Voltages (High and Low) to set the thresholds for the logic levels of the pin designated. (AD)

The data characteristics for I/O Pins are stored in an array of *IOCFG* structs called *IO_Pins*. It includes the idle state of the pin, the polarity of the pin, if the pin uses differential signaling, and the pin data state.

```
typedef struct {
  IOCFG_IDLE_STATE idleState; ///< Whether the pin is high or low when idling
  IOCFG_POLARITY polarity; ///< Whether the pin output bits are flipped
  IOCFG_DIFFERENTIAL differential; ///< If the differential pair is setup
  IOCFG_DATA_STATE dataState; ///< Whether the pin is an input, output, or clock
} IOCFG;

IOCFG IO_Pins[NUM_IO_PINS];
```

An idle state can either be set to an idle high signal, an idle low signal, or in tri-state. The polarity can be either true or false, which will flip each data bit before sending or after receiving data. Differential signaling can be either enabled or disabled.

```
typedef enum {
  IOCFG_IDLE_STATE_HIGH,
  IOCFG_IDLE_STATE_LOW,
  IOCFG_IDLE_STATE_TRISTATE,
} IOCFG_IDLE_STATE;

typedef enum {
  IOCFG_POLARITY_TRUE,
  IOCFG_POLARITY_FALSE,
} IOCFG_POLARITY;

typedef enum {
  IOCFG_DIFFERENTIAL_ENABLED,
  IOCFG_DIFFERENTIAL_DISABLED,
} IOCFG_DIFFERENTIAL;
```

The data state of an I/O pin by default is disabled. Each pin can be configured as an input, output, clock, or a chip select pin.

```
typedef enum {
  IOCFG_DATA_STATE_DISABLED,
  IOCFG_DATA_STATE_INPUT,
  IOCFG_DATA_STATE_OUTPUT,
  IOCFG_DATA_STATE_CLOCK,
  IOCFG_DATA_STATE_CS,
} IOCFG_DATA_STATE;
```

To make checking if a pin number is valid easier, a macro is defined to automatically return *IOM_ERROR_INVALID* if a given pin number is not in the range.

```
#define CHECK_PIN(pin) if ((pin == 0) || (pin > NUM_IO_PINS)) \
                              return IOM_ERROR_INVALID
```

SetIOPinIdleState takes a pin number and the new idle state and updates the output port to the new state, along with updating it in the IO_Pins array.

```
IOM_ERROR SetIOPinIdleState(size_t pinNumber, IOCFG_IDLE_STATE idleState) {
  CHECK_PIN(pinNumber);
  if (idleState == IO_Pins[pinNumber-1].idleState) {
     //Nothing to do, Idle state already set
     return IOM_OK;
  }
  uint16_t ioPin = 0;
  switch(pinNumber) {
     case 1:
     ioPin = IO_1_OUT_Pin;
     break;
     case 2:
     ioPin = IO_2_OUT_Pin;
     break;
     case 3:
     ioPin = IO_3_OUT_Pin;
     break;
     case 4:
     ioPin = IO_4_OUT_Pin;
     break;
  }
  switch(idleState) {
     case IOCFG_IDLE_STATE_HIGH:
     IO_PIN_GPIO_OUTPUT_PORT->ODR |= (ioPin << 1);
     break;
     case IOCFG_IDLE_STATE_LOW:
     IO_PIN_GPIO_OUTPUT_PORT->ODR &= ~((ioPin << 1));
     break;
     case IOCFG_IDLE_STATE_TRISTATE:
     break;
  }
  IO_Pins[pinNumber-1].idleState =  idleState;
  return IOM_OK;
}
```

SetIOPinPolarity updates the *IO_Pins* array with the new polarity option.

```
IOM_ERROR SetIOPinPolarity(size_t pinNumber, IOCFG_POLARITY polarity) {
  CHECK_PIN(pinNumber);
  IO_Pins[pinNumber-1].polarity = polarity;
  return IOM_OK;
}
```

EnableDifferentialMode and *DisableDifferentialMode* control the differential mode for the pins.

```
IOM_ERROR EnableDifferentialMode(size_t pinNumber) {
```

```
  CHECK_PIN(pinNumber);
  IO_Pins[pinNumber-1].differential = IOCFG_DIFFERENTIAL_ENABLED;
  return IOM_OK;
}

IOM_ERROR DisableDifferentialMode(size_t pinNumber) {
  CHECK_PIN(pinNumber);
  IO_Pins[pinNumber-1].differential = IOCFG_DIFFERENTIAL_DISABLED;
  return IOM_OK;
}
```

The command for setting the data state is processed for all of the pins. Then each pin calls the *SetIOPinDataState* function to make the changes necessary.

```
IOM_ERROR SetPinStates(uint8_t* comm) {
     uint8_t pin1State = *(comm+1) & 0b11;
     uint8_t pin2State = (*(comm+1) & 0b1100) >> 2;
     uint8_t pin3State = (*(comm+1) & 0b110000) >> 4;
     uint8_t pin4State = (*(comm+1) & 0b110000) >> 6;
     IOCFG_DATA_STATE dataState = IOCFG_DATA_STATE_DISABLED;
     IOM_ERROR err;
     err = GetDataState(pin1State, &dataState);
     if (err != IOM_OK) {
     return err;
     }
     SetIOPinDataState(pin1State, dataState);

     err = GetDataState(pin2State, &dataState);
     if (err != IOM_OK) {
     return err;
     }
     SetIOPinDataState(pin2State, dataState);

     err = GetDataState(pin3State, &dataState);
     if (err != IOM_OK) {
     return err;
     }
     SetIOPinDataState(pin3State, dataState);

     err = GetDataState(pin4State, &dataState);
     if (err != IOM_OK) {
     return err;
     }
     SetIOPinDataState(pin4State, dataState);

     return IOM_OK;
}
```

SetIOPinDataState controls if the pin is an input, output, clock, chip select, or disabled. If the new state is the same as the old one, nothing changes.

```
IOM_ERROR SetIOPinDataState(size_t pinNumber, IOCFG_DATA_STATE dataState) {
  CHECK_PIN(pinNumber);
  //If the pin already was configured for this, return IOM_OK
  if (IO_Pins[pinNumber-1].dataState == dataState) {
     return IOM_OK;
  }
  IOCFG_DATA_STATE lastState = IO_Pins[pinNumber-1].dataState;
  IO_Pins[pinNumber-1].dataState = dataState;
  switch(dataState) {
```

If the new datastate is an output or chip select, the clock pin will be disabled if the pin used to be a clock. Then, the output pin is enabled.

```
     case IOCFG_DATA_STATE_OUTPUT:
     case IOCFG_DATA_STATE_CS:
     if (lastState == IOCFG_DATA_STATE_CLOCK) {
     DisableClockPin(pinNumber);
     }
     EnableOutputPin(pinNumber);
     break;
```

DisableClockPin will find the corresponding GPIO port and number for the pin, then de-initialize the pin. The pin is then reinitialized as an input.

```
IOM_ERROR DisableClockPin(uint8_t pinNumber) {
  CHECK_PIN(pinNumber);
  GPIO_TypeDef* clkPort = 0;
  uint16_t clkPin = 0;
  switch(pinNumber) {
     case 1:
     clkPort = IO_1_CLK_GPIO_Port;
     clkPin = IO_1_CLK_Pin;
     break;
     case 2:
     clkPort = IO_2_CLK_GPIO_Port;
     clkPin = IO_2_CLK_Pin;
     break;
     case 3:
     clkPort = IO_3_CLK_GPIO_Port;
     clkPin = IO_3_CLK_Pin;
     break;
```

```
      case 4:
      clkPort = IO_4_CLK_GPIO_Port;
      clkPin = IO_4_CLK_Pin;
      break;
   }
   HAL_GPIO_DeInit(clkPort, clkPin);
   GPIO_InitTypeDef GPIO_InitStruct = {0};
   GPIO_InitStruct.Pin = clkPin;
   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
   GPIO_InitStruct.Pull = GPIO_NOPULL;
   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
   HAL_GPIO_Init(clkPort, &GPIO_InitStruct);

   return IOM_OK;

 }
```

EnableOutputPin will find the corresponding GPIO port and number for the pin, then deinitialize the pin. Then, the pin's output is set to its current idle state. Afterwards, the pin is enabled as an output.

```
 IOM_ERROR EnableOutputPin(uint8_t pinNumber) {
   CHECK_PIN(pinNumber);
   GPIO_TypeDef* outPort = 0;
   uint16_t outPin = 0;
   switch(pinNumber) {
      case 1:
      outPort = IO_1_OUT_GPIO_Port;
      outPin = IO_1_OUT_Pin;
      break;
      case 2:
      outPort = IO_2_OUT_GPIO_Port;
      outPin = IO_2_OUT_Pin;
      break;
      case 3:
      outPort = IO_3_OUT_GPIO_Port;
      outPin = IO_3_OUT_Pin;
      break;
      case 4:
      outPort = IO_4_OUT_GPIO_Port;
      outPin = IO_4_OUT_Pin;
   }
   HAL_GPIO_DeInit(outPort, outPin);
   switch(IO_Pins[pinNumber].idleState) {
      case IOCFG_IDLE_STATE_HIGH:
      IO_PIN_GPIO_OUTPUT_PORT->ODR |= (outPin << 1);
      break;
      case IOCFG_IDLE_STATE_LOW:
```

```
      IO_PIN_GPIO_OUTPUT_PORT->ODR &= ~((outPin << 1));
      break;
      case IOCFG_IDLE_STATE_TRISTATE:
      break;
   }
   GPIO_InitTypeDef GPIO_InitStruct = {0};
   GPIO_InitStruct.Pin = outPin;
   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
   GPIO_InitStruct.Pull = GPIO_NOPULL;
   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
   HAL_GPIO_Init(outPort, &GPIO_InitStruct);
   return IOM_OK;

 }
```

If the new state of the pin is a clock, the output pin will be disabled if applicable, and the clock pin is enabled.

```
      case IOCFG_DATA_STATE_CLOCK:
      if (lastState == IOCFG_DATA_STATE_OUTPUT ||
            lastState == IOCFG_DATA_STATE_CS) {
            DisableOutputPin(pinNumber);
      }
      EnableClockPin(pinNumber);
      break;
```

DisableOutputPin will find the corresponding GPIO port and number for the pin, then deinitialize the pin. The pin is then reinitialized as an input.

```
IOM_ERROR DisableOutputPin(uint8_t pinNumber) {
  CHECK_PIN(pinNumber);
  GPIO_TypeDef* outPort = 0;
  uint16_t outPin = 0;
  switch(pinNumber) {
     case 1:
     outPort = IO_1_OUT_GPIO_Port;
     outPin = IO_1_OUT_Pin;
     break;
     case 2:
     outPort = IO_2_OUT_GPIO_Port;
     outPin = IO_2_OUT_Pin;
     break;
     case 3:
     outPort = IO_3_OUT_GPIO_Port;
     outPin = IO_3_OUT_Pin;
     break;
```

```
      case 4:
      outPort = IO_4_OUT_GPIO_Port;
      outPin = IO_4_OUT_Pin;
   }
   HAL_GPIO_DeInit(outPort, outPin);
   GPIO_InitTypeDef GPIO_InitStruct = {0};
   GPIO_InitStruct.Pin = outPin;
   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
   GPIO_InitStruct.Pull = GPIO_NOPULL;
   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
   HAL_GPIO_Init(outPort, &GPIO_InitStruct);
   return IOM_OK;

 }
```

EnableClockPin has more steps than the other enable/disable functions. Along with determining the port and pin number for the clock, the timer and channel also is determined based on the pin number. Then, the timer is enabled and the period is set. The clock pin is then deinitialized and reinitialized as a clock.

```
IOM_ERROR EnableClockPin(uint8_t pinNumber) {
  CHECK_PIN(pinNumber);
  GPIO_TypeDef* clkPort = 0;
  uint16_t clkPin = 0;
  uint32_t alternate = 0;
  TIM_TypeDef* pTimer = 0;
  uint32_t channel = 0;
  switch(pinNumber) {
     case 1:
     clkPort = IO_1_CLK_GPIO_Port;
     clkPin = IO_1_CLK_Pin;
     alternate = GPIO_AF1_TIM2;
     pTimer = TIM2;
     channel = TIM_CCER_CC1E;
     break;
     case 2:
     clkPort = IO_2_CLK_GPIO_Port;
     clkPin = IO_2_CLK_Pin;
     alternate = GPIO_AF2_TIM5;
     pTimer = TIM5;
     channel = TIM_CCER_CC2E;

     break;
     case 3:
     clkPort = IO_3_CLK_GPIO_Port;
     clkPin = IO_3_CLK_Pin;
     alternate = GPIO_AF4_TIM15;
```

```
      pTimer = TIM15;
      channel = TIM_CCER_CC1E;
      break;
      case 4:
      clkPort = IO_4_CLK_GPIO_Port;
      clkPin = IO_4_CLK_Pin;
      alternate = GPIO_AF2_TIM3;
      pTimer = TIM3;
      channel = TIM_CCER_CC1E;
      break;
    }

    pTimer->CCER |= channel;
    pTimer->BDTR |= TIM_BDTR_MOE;
    pTimer->CNT = timerPeriod;
    pTimer->CR1 |= TIM_CR1_CEN;
    HAL_GPIO_DeInit(clkPort, clkPin);
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    GPIO_InitStruct.Pin = clkPin;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    GPIO_InitStruct.Alternate = alternate;
    HAL_GPIO_Init(clkPort, &GPIO_InitStruct);

    return IOM_OK;
  }
```

If the new data state is an input or disabled, then either the clock pin or output pin are disabled, depending on which state the pin was last in. All disabled pins are set up as inputs as input pins draw less power than floating pins. (CS)

```
      case IOCFG_DATA_STATE_INPUT:
      case IOCFG_DATA_STATE_DISABLED:
      if (lastState == IOCFG_DATA_STATE_CLOCK) {
      DisableClockPin(pinNumber);
      } else {
      DisableOutputPin(pinNumber);
      }
      break;
    }
    return IOM_OK;
  }
```

Table 53: Data Commands

Byte 1	Byte 2	Byte 3	Byte 4-N	Command Name
0x03	0x00	numBytes	Data...	Set Data to Send
0x03	0x01	numBytes	N/A	Set amount of Data to receive

Table 53 lists the currently designed data commands for the I/O Master. The first command is *Set Data to Send*. This command is the only command that can vary in length. Byte 3 contains the number of bytes of data to send, so the command must be parsed dynamically (the length cannot be assumed). The other command, *Set amount of Data to receive*, is used to configure the I/O master to read in a specified number of bytes.

Since the *Set Data to Send* command has a dynamic length, the processing of these commands is more complex than the others. For this reason, the design of processing commands was being reworked to be simpler and allow continued processing while more data became available simultaneously, but this was not fully implemented before work was halted. The original implementation uses a method of tracking the identification of the command while it is being processed.

```
void ProcessDataCommand(IOM_Output_Buffer buffer) {
     uint8_t* pData;
     switch(command_status) {
```

At first, the command is typed as *IOM_CS_NEW*, which is the state of a new, unknown command. As the command is processed, the command either gets classified as *IOM_CS_DATA_INC* which is an incompletely known data command, or *IOM_CS_DATA*, which

is a completely known data command. If the command is not either of those, the data is thrown out and the I/O Master waits for the next command.

```
case IOM_CS_NEW:
	current_command[1] = *(buffer.data + 1);
	free(buffer.data);
	switch(current_command[1]) {
		case 1:
		command_status = IOM_CS_DATA_INC;
		pData = malloc(1);
		UARTQueueRXData(pData, 1);
		return;
		case 2:
		command_status = IOM_CS_DATA;
		pData = malloc(1);
		UARTQueueRXData(pData, 1);
		return;
		default:
		//Garbage
		pData = malloc(2);
		UARTQueueRXData(pData, 2);
		return;
	}
	break;
```

If the command type is *IOM_CS_DATA_INC*, then the command must be a load data command. The length of the command is confirmed, and then the next byte is fetched.

```
case IOM_CS_DATA_INC:
	if (buffer.length != 1) {
		//Garbage
		free(buffer.data);
		pData = malloc(2);
		UARTQueueRXData(pData, 2);
		return;
	}
	command_status = IOM_CS_DATA_LOAD;
	uint8_t size = *(buffer.data);
	free(buffer.data);
	pData = malloc(size);
	UARTQueueRXData(pData,size);
	return;
```

If the command type is *IOM_CS_DATA*, then this is a command to specify how many bytes of data to receive. The information is set on the I/O Master and it waits for the next command to process. (CS)

```
	case IOM_CS_DATA:
		if (buffer.length != 1) {
			//Garbage
			free(buffer.data);
			pData = malloc(2);
			UARTQueueRXData(pData, 2);
		}
		command_status = IOM_CS_NEW;
		dataToReceive = *(buffer.data);
		free(buffer.data);
		pData = malloc(2);
		UARTQueueRXData(pData, 2);
		return;
```

Finally, if the command type is IOM_CS_DATA_LOAD, then all of the data is available from the command.

```
	case IOM_CS_DATA_LOAD:
		command_status = IOM_CS_NEW;
		QueueOutputDataToSend(buffer.data, buffer.length, 3);
		free(buffer.data);
		pData = malloc(2);
		UARTQueueRXData(pData, 2);
		return;
	default:
		//Garbage
		command_status = IOM_CS_NEW;
		free(buffer.data);
		pData = malloc(2);
		UARTQueueRXData(pData, 2);
		return;
	}
}
```

On power up the I/O Master is configured to a default state. In order to change this, a configuration command needs to be received. The configuration command specifies which of the following settings need adjusted and the desired state in which to configure them. There are three

types of outputs needed to configure the Level Shifter: MCU GPIO, MCU DACs, and I^2C Commands.

MCU GPIO is set up by assigning pins on the STM32 as GPIO_Output within STM32CubeMX and then assigning them a user label. Once this is configured the pin voltage can be toggled between 0V and 3.3V by using the HAL GPIO APIs. Example commands are shown below.

```
HAL_GPIO_TogglePin(GPIOB, LD1_Pin);//Toggle LD1
HAL_GPIO_WritePin(GPIOB, LD1_Pin, GPIO_PIN_SET);// Set LD1 HI
HAL_GPIO_WritePin(GPIOB, LD1_Pin, GPIO_PIN_RESET);// Set LD1 LO
```

The following are configured by setting their respective MCU Pin HI(3.3V) or LO(0V): Level Shifter Signalling Mode, I/O Pin Pull-up Resistor, I/O Pin Pull-down Resistor, I/O Pin Termination Resistor.

MCU DACs are set up by assigning pins PA4 and PA5 as DAC1_OUT1 and DAC1_OUT2 respectively within STM32CubeMX. This configures the pins to be 2 channels of the DAC built into the MCU. To begin using these channels, they need to be enabled individually using the HAL API commands.

```
HAL_DAC_Start(&hdac1, DAC1_CHANNEL_1);
HAL_DAC_Start(&hdac1, DAC1_CHANNEL_2);
```

Then the DAC output can be configured using

```
HAL_DAC_SetValue(&hdac1, DAC1_CHANNEL_1, DAC_ALIGN_12B_R, DAC_VALUE);
HAL_DAC_SetValue(&hdac1, DAC1_CHANNEL_2, DAC_ALIGN_12B_R, DAC_VALUE);
```

MCU DACs are set up by assigning pins PA4 and PA5 as DAC1_OUT1 and DAC1_OUT2 respectively within STM32CubeMX. This configures the pins to be two channels of the DAC

built into the MCU. To begin using these channels, each needs to be enabled individually using the Where DAC_VALUE is a 12 bit integer. This number is scaled to the range 0 to 3.3V on the output such that

$$V_{DAC} = \frac{3.3}{2^{12}} \times DAC\_VALUE$$

DAC channel one is used to control the target device power output. This is done by applying a voltage to the regulator's feedback loop in order to augment the voltage divider and therefore adjust the voltage output. DAC channel two is unused.

```
void writeMcuDac(uint channel, float Vout)
{
  if(channel == 1)
  {
    channel = DAC1_CHANNEL_1;
  }
  elseif(channel == 2)
  {
    channel = DAC1_CHANNEL_2;
  }
  uint16_t DAC_VALUE = round(Vout*4095/3.3);
  //Write Channel Register
  HAL_DAC_SetValue(&hdac1, channel, DAC_ALIGN_12B_R, DAC_VALUE);
}
```

I^2C is setup by assigning "I2C#_SCL" and "I2C#_SDA" to a pair of pins, where # is a numeral one to four. These pins are connected to the SCL and SDA lines of slave device(s). Commands can then be sent using:

```
HAL_I2C_Master_Transmit(&hi2c2, ADDR, CommandPtr, numBytes, HAL_MAX_DELAY);
```

Where ADDR is the address of the slave device, CommandPtr is a pointer to the byte array of the command, and numBytes is the number of bytes in the command.

The I/O Master only needs I^2C to control additional DACs (MCP4728) which generate the logic-high voltage, VH, and the logic-low voltage, VL. The MCP4728 has four DAC channels, each with a 12 bit buffer to control the voltage output. To change the voltage output, three commands must be sent. First the VRef is set during initialization (see InitI2CDAC()). Then the DAC buffer is written. Then an update command is sent. The code below shows how this is done using the HAL I2C APIs. Note that the DAC itself outputs 0 to 3.3V, but each output has an op-amp buffer circuit which scales the voltage to the range of -15 to 15V which is the expected value for Vout. Thus the value written to the buffer is calculated using the following:

$$DAC\_VALUE = \frac{2^{12} - 1}{30} \times (15 - V_{OUT})$$

Thus the following functions can be used to adjust the desired frequency using the HAL DAC APIs. (AD)

```
uint8_t IO_DAC_UPDATE_BUF = 0x08; //General Call Software Update
uint8_t io_dac_buf[16] = {0}; //Buffer for all VH/VL values


  IOM_ERROR WriteExtDAC(IOM_REGISTER reg, uint8_t channel, float Vout)
{
  //HAL_StatusTypeDef ret;
  uint16_t dacVal = round(136.5 * (15 - Vout));//((2^12 - 1) / 30) * (15 - Vout)
  io_dac_buf[(channel - 1) * 4 + 2*reg] = (dacVal >> 8) & 0x00FF;
  io_dac_buf[(channel - 1) * 4 + 2*reg + 1] = dacVal & 0x00FF;

  uint8_t *buf;
  uint16_t addr;
  switch(channel) {
    case 1:
    case 2:
      buf = io_dac_buf;
      addr = IO_DAC_CH_1_2_ADDR;
      break;
    case 3:
    case 4:
      buf = io_dac_buf + 8; //+8 because 2 channels at same time
      addr = IO_DAC_CH_3_4_ADDR;
```

```
        break;
      default:
        return IOM_ERROR_INVALID;
    }


    //Write Channel Registers
    HAL_I2C_Master_Transmit(&hi2c3, addr, buf, 8, HAL_MAX_DELAY);

    //Update Voltages
    HAL_I2C_Master_Transmit(&hi2c3, IO_DAC_UPDATE_ADDR, &IO_DAC_UPDATE_BUF, 1,
HAL_MAX_DELAY);
    return IOM_OK;
  }
```

The following are the implementations of all the configuration commands.

RunConfigurationCommand() takes a byte array containing the configuration command being processed and calls the appropriate function to perform it.

```
    void RunConfigurationCommand(uint8_t* comm)
    {
        uint8_t readWrite = (*(comm) & 0b10000000) >> 7;
        uint8_t commTag = (*(comm) & 0b01110000) >> 4;

        if(readWrite != 0)
        {
            //Set cases
            switch(commTag)
            {
                case 0:
                    SetTargetDeviceVoltage(comm);
                    break;
                case pin1Params:
                case pin2Params:
                case pin3Params:
                case pin4Params:
                    SetPinParams(commTag, comm);
                    break;
                case dataSpeed:
                    SetDataSpeed(comm);
                    break;
                case signalMode:
                    SetSignalMode(comm);
                    break;
                case pinStates:
                    SetPinStates(comm);
                default:
                    //error case
```

```
                break;
            }
        }
        else
        {
            //Get cases
            switch(commTag)
            {
                case targetDeviceVolt:
                    GetTargetDeviceVoltage(comm);
                    break;
                case pin1Params:
                case pin2Params:
                case pin3Params:
                case pin4Params:
                    GetPinParams(commTag, comm);
                    break;
                case dataSpeed:
                    GetDataSpeed(comm);
                    break;
                case signalMode:
                    GetSignalMode(comm);
                    break;
                default:
                    //error case
                    break;
            }
        }

        return;
    }

    IOM_ERROR SetPinParams(uint8_t pinNum, uint8_t* comm)
    {
        uint8_t pullDown = comm[0] & 0b0001;
        uint8_t pullUp = (comm[0] & 0b0010) >> 1;
        //If pullDown and pullUp are enabled, then board is in tristate mode
        uint8_t triState = pullDown & pullUp;
        uint16_t VH = (comm[1]<<8) | comm[2];
        uint16_t VL = (comm[3]<<8) | comm[4];

        GPIO_TypeDef* pullDownPort = 0;
        uint16_t pullDownPin = 0;
        GPIO_TypeDef* pullUpPort = 0;
        uint16_t pullUpPin = 0;

        GPIO_TypeDef* triStatePort = 0;
        uint16_t triStatePin = 0;

        switch(pinNum) {
            case 1:
                pullDownPort = IO_1_PD_N_GPIO_Port;
                pullDownPin = IO_1_PD_N_Pin;
```

```
            pullUpPort = IO_1_PU_GPIO_Port;
            pullUpPin = IO_1_PU_Pin;
            triStatePort = IO_1_TRIS_N_GPIO_Port;
            triStatePin = IO_1_TRIS_N_Pin;
            break;
        case 2:
            pullDownPort = IO_2_PD_N_GPIO_Port;
            pullDownPin = IO_2_PD_N_Pin;
            pullUpPort = IO_2_PU_GPIO_Port;
            pullUpPin = IO_2_PU_Pin;
            triStatePort = IO_2_TRIS_N_GPIO_Port;
            triStatePin = IO_2_TRIS_N_Pin;
            break;
        case 3:
            pullDownPort = IO_3_PD_N_GPIO_Port;
            pullDownPin = IO_3_PD_N_Pin;
            pullUpPort = IO_3_PU_GPIO_Port;
            pullUpPin = IO_3_PU_Pin;
            triStatePort = IO_3_TRIS_N_GPIO_Port;
            triStatePin = IO_3_TRIS_N_Pin;
            break;
        case 4:
            pullDownPort = IO_4_PD_N_GPIO_Port;
            pullDownPin = IO_4_PD_N_Pin;
            pullUpPort = IO_4_PU_GPIO_Port;
            pullUpPin = IO_4_PU_Pin;
            triStatePort = IO_4_TRIS_N_GPIO_Port;
            triStatePin = IO_4_TRIS_N_Pin;
            break;
        default:
            return IOM_ERROR_INVALID;
    }

    if (triState != 0) {
        //Dsiable the Pulldown and Pullup Pins
        HAL_GPIO_WritePin(pullDownPort, pullDownPin, GPIO_PIN_SET);
        HAL_GPIO_WritePin(pullUpPort, pullUpPin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(triStatePort, triStatePin, GPIO_PIN_RESET);

    } else if (pullDown != 0) {
        HAL_GPIO_WritePin(triStatePort, triStatePin, GPIO_PIN_SET);
        HAL_GPIO_WritePin(pullUpPort, pullUpPin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(pullDownPort, pullDownPin, GPIO_PIN_RESET);

    } else if (pullUp != 0) {
        HAL_GPIO_WritePin(triStatePort, triStatePin, GPIO_PIN_SET);
        HAL_GPIO_WritePin(pullDownPort, pullDownPin, GPIO_PIN_SET);
        HAL_GPIO_WritePin(pullUpPort, pullUpPin, GPIO_PIN_SET);
    } else if (pullDown == 0 && pullUp == 0 && triState == 0) {
        HAL_GPIO_WritePin(triStatePort, triStatePin, GPIO_PIN_SET);
        HAL_GPIO_WritePin(pullDownPort, pullDownPin, GPIO_PIN_SET);
        HAL_GPIO_WritePin(pullUpPort, pullUpPin, GPIO_PIN_RESET);
    } else {
```

```
            return IOM_ERROR_INVALID;
        }

        WriteRegExtDAC(IOM_REGISTER_HIGH, pinNum, VH);
        WriteRegExtDAC(IOM_REGISTER_LOW, pinNum, VL);
        return IOM_OK;

    }

    void SetTargetDeviceVoltage(uint8_t* comm)
    {
        uint16_t voltage = (comm[1]<<8) | comm[2];
        WriteMCUDAC(1, voltage);
    }

    void SetSignalMode(uint8_t* comm)
    {
        uint8_t pin12Mode = comm[0] & 0x01;
        uint8_t pin34Mode = (comm[0] >> 1) & 0x01;
        #ifdef IOM_8_IO_PINS
        uint8_t pin56Mode = (comm[0] >> 2) & 0x01;
        uint8_t pin78Mode = (comm[0] >> 3) & 0x01;
        #endif

        HAL_GPIO_WritePin(IO_1_2_DIFF_GPIO_Port, IO_1_2_DIFF_Pin, (pin12Mode != 0)
? GPIO_PIN_SET : GPIO_PIN_RESET);
        HAL_GPIO_WritePin(IO_3_4_DIFF_GPIO_Port, IO_3_4_DIFF_Pin, (pin34Mode != 0)
? GPIO_PIN_SET : GPIO_PIN_RESET);
        #ifdef IOM_8_IO_PINS
        HAL_GPIO_WritePin(IO_5_6_DIFF_GPIO_Port, IO_5_6_DIFF_Pin, (pin56Mode != 0)
? GPIO_PIN_SET : GPIO_PIN_RESET);
        HAL_GPIO_WritePin(IO_7_8_DIFF_GPIO_Port, IO_7_8_DIFF_Pin, (pin78Mode != 0)
? GPIO_PIN_SET : GPIO_PIN_RESET);
        #endif

        HAL_GPIO_WritePin(IO_1_2_TERM_N_GPIO_Port, IO_1_2_TERM_N_Pin, (pin12Mode ==
0) ? GPIO_PIN_SET : GPIO_PIN_RESET);
        HAL_GPIO_WritePin(IO_3_4_TERM_N_GPIO_Port, IO_3_4_TERM_N_Pin, (pin34Mode ==
0) ? GPIO_PIN_SET : GPIO_PIN_RESET);
        #ifdef IOM_8_IO_PINS
        HAL_GPIO_WritePin(IO_5_6_TERM_N_GPIO_Port, IO_5_6_TERM_N_Pin, (pin56Mode ==
0) ? GPIO_PIN_SET : GPIO_PIN_RESET);
        HAL_GPIO_WritePin(IO_7_8_TERM_N_GPIO_Port, IO_7_8_TERM_N_Pin, (pin78Mode ==
0) ? GPIO_PIN_SET : GPIO_PIN_RESET);
        #endif

        return;
    }

    void SetDataSpeed(uint8_t* comm)
    {
        //uint speed;//TODO: Update when Set Data Speed Commmand established
```

```
    //return;
}

IOM_ERROR SetPinStates(uint8_t* comm) {
    uint8_t pin1State = *(comm+1) & 0b11;
    uint8_t pin2State = (*(comm+1) & 0b1100) >> 2;
    uint8_t pin3State = (*(comm+1) & 0b110000) >> 4;
    uint8_t pin4State = (*(comm+1) & 0b110000) >> 6;
    IOCFG_DATA_STATE dataState = IOCFG_DATA_STATE_DISABLED;
    IOM_ERROR err;
    err = GetDataState(pin1State, &dataState);
    if (err != IOM_OK) {
        return err;
    }
    SetIOPinDataState(pin1State, dataState);

    err = GetDataState(pin2State, &dataState);
    if (err != IOM_OK) {
        return err;
    }
    SetIOPinDataState(pin2State, dataState);

    err = GetDataState(pin3State, &dataState);
    if (err != IOM_OK) {
        return err;
    }
    SetIOPinDataState(pin3State, dataState);

    err = GetDataState(pin4State, &dataState);
    if (err != IOM_OK) {
        return err;
    }
    SetIOPinDataState(pin4State, dataState);

    return IOM_OK;

}

IOM_ERROR GetPinParams(uint8_t pinNum, uint8_t* comm)
{
    GPIO_TypeDef* pullDownPort = 0;
    uint16_t pullDownPin = 0;
    GPIO_TypeDef* pullUpPort = 0;
    uint16_t pullUpPin = 0;

    GPIO_TypeDef* triStatePort = 0;
    uint16_t triStatePin = 0;

    switch(pinNum) {
        case 1:
            pullDownPort = IO_1_PD_N_GPIO_Port;
            pullDownPin = IO_1_PD_N_Pin;
            pullUpPort = IO_1_PU_GPIO_Port;
```

```
                pullUpPin = IO_1_PU_Pin;
                triStatePort = IO_1_TRIS_N_GPIO_Port;
                triStatePin = IO_1_TRIS_N_Pin;
                break;
            case 2:
                pullDownPort = IO_2_PD_N_GPIO_Port;
                pullDownPin = IO_2_PD_N_Pin;
                pullUpPort = IO_2_PU_GPIO_Port;
                pullUpPin = IO_2_PU_Pin;
                triStatePort = IO_2_TRIS_N_GPIO_Port;
                triStatePin = IO_2_TRIS_N_Pin;
                break;
            case 3:
                pullDownPort = IO_3_PD_N_GPIO_Port;
                pullDownPin = IO_3_PD_N_Pin;
                pullUpPort = IO_3_PU_GPIO_Port;
                pullUpPin = IO_3_PU_Pin;
                triStatePort = IO_3_TRIS_N_GPIO_Port;
                triStatePin = IO_3_TRIS_N_Pin;
                break;
            case 4:
                pullDownPort = IO_4_PD_N_GPIO_Port;
                pullDownPin = IO_4_PD_N_Pin;
                pullUpPort = IO_4_PU_GPIO_Port;
                pullUpPin = IO_4_PU_Pin;
                triStatePort = IO_4_TRIS_N_GPIO_Port;
                triStatePin = IO_4_TRIS_N_Pin;
                break;
            default:
                return IOM_ERROR_INVALID;
        }
        uint8_t pullUp = (HAL_GPIO_ReadPin(pullUpPort, pullUpPin) == GPIO_PIN_SET)
? 1 : 0;
        uint8_t pullDown = (HAL_GPIO_ReadPin(pullDownPort, pullDownPin) ==
GPIO_PIN_SET) ? 1 : 0;
        uint8_t triState = (HAL_GPIO_ReadPin(triStatePort, triStatePin) ==
GPIO_PIN_SET) ? 1 : 0;

        *comm = pinNum << 4 & 0x30;
        if (pullDown != 0) {
            *comm |= 0x01;
        } else if (pullUp!= 0) {
            *comm |= 0x02;
        } else if (triState != 0) {
            *comm |= 0x03;
        }

        //get VH
        uint16_t* pVComm = (uint16_t*)(comm+1);
        *pVComm = ReadExtDAC(IOM_REGISTER_HIGH, pinNum);
        pVComm = (uint16_t*)(comm+3);
        *pVComm = ReadExtDAC(IOM_REGISTER_LOW, pinNum);
```

```
        return IOM_OK;
    }

    void GetTargetDeviceVoltage(uint8_t* comm)
    {
        //device volt is on Dac channel 1
        // last 12 bits are the channel 1 output
        //uint16_t Volt = HAL_DAC_GetValue(&hdac1, DAC1_CHANNEL_1); // & 0x00000FFF
        //return;
    }

    void GetSignalMode(uint8_t* comm)
    {
        //comm[0] = comm[0] & HAL_GPIO_ReadPin(IO_1_2_DIFF_GPIO_Port,
IO_1_2_DIFF_Pin);
        //return;
    }

    void GetDataSpeed(uint8_t* comm)
    {
        //uint8_t speed;//TODO: Update when Get Data Speed Commmand established
        //return;
    }
```

Most of the testing on the code was done during implementation of the board configurator code. Tests included setting every pin to an input, output, clock, and chip select to ensure that all of the pins could be reconfigured dynamically. In addition, the entire logic level range for each pin was tested and confirmed to be functional. The I/O Master was connected to an oscilloscope to monitor the voltage levels on each I/O pin.

5.3.9 I/O Data Buffer

To read and write data with the I/O Master, the on board microcontroller uses GPIO pins. By using general purpose pins, the same pins can be configured to operate with all of the target protocols (I^2C, SPI, UART, RS232, RS485, SWD), and the design offers flexibility to implement additional protocols in the future without a significant amount of work. To match the need and want statements for the I/O Master, such a general purpose design is necessary.

These GPIO pins are tied to memory using DMA. The DMA trigger can be synchronized to an onboard timer, meaning the speed of data input and output can be controlled without needing to utilize CPU clock cycles. Table 54 shows the configuration used by the I/O Master to write data from memory to GPIO using DMA. Table 55 shows a similar configuration, but for reading data from GPIO to memory.

Table 54: Configuration Options for DMA Controller for Writing Data

DMA Controller Option	**Value**
DMA Stream	DMA 2 Stream 7
Request Generator	Timer 8 Channel 4
Stream Direction	Memory to Peripheral
Memory address increment mode	Enabled
Peripheral address increment mode	Disabled
Memory Data Alignment	1 Byte
Peripheral Data Alignment	1 Byte
DMA Mode	Normal mode
DMA Priority	Very High
FIFO Mode	Enabled
FIFO Threshold	1 Quarter Full
Memory Burst	Single Burst
Peripheral Burst	Single Burst

Table 55: Configuration Options for DMA Controller for Reading Data

DMA Controller Option	**Value**
DMA Stream	DMA 2 Stream 8
Request Generator	Timer 8 Channel 3
Stream Direction	Peripheral to Memory
Memory address increment mode	Enabled
Peripheral address increment mode	Disabled
Memory Data Alignment	1 Byte
Peripheral Data Alignment	1 Byte
DMA Mode	Circular Mode
DMA Priority	Very High
FIFO Mode	Disabled
Memory Burst	Single Burst
Peripheral Burst	Single Burst

The DMA controller has several configuration settings to consider. The DMA request generator needs to be tied to an advanced timer channel so that the timer can trigger sending data over DMA. The direction of the DMA stream must be defined as memory to peripheral (writing data) or peripheral to memory (reading data). It is crucial to ensure that the memory address pointer is set to increment each time the DMA is triggered, but that the peripheral address pointer is not. The minimum amount of data that can be transferred using DMA is one byte, so one byte is configured as the data block size. Additional options vary depending on if DMA is being used to read or write data. Figure 62 depicts the route configured to stream data from memory in domain D2 to GPIO on the AHB4 bus using DMA.

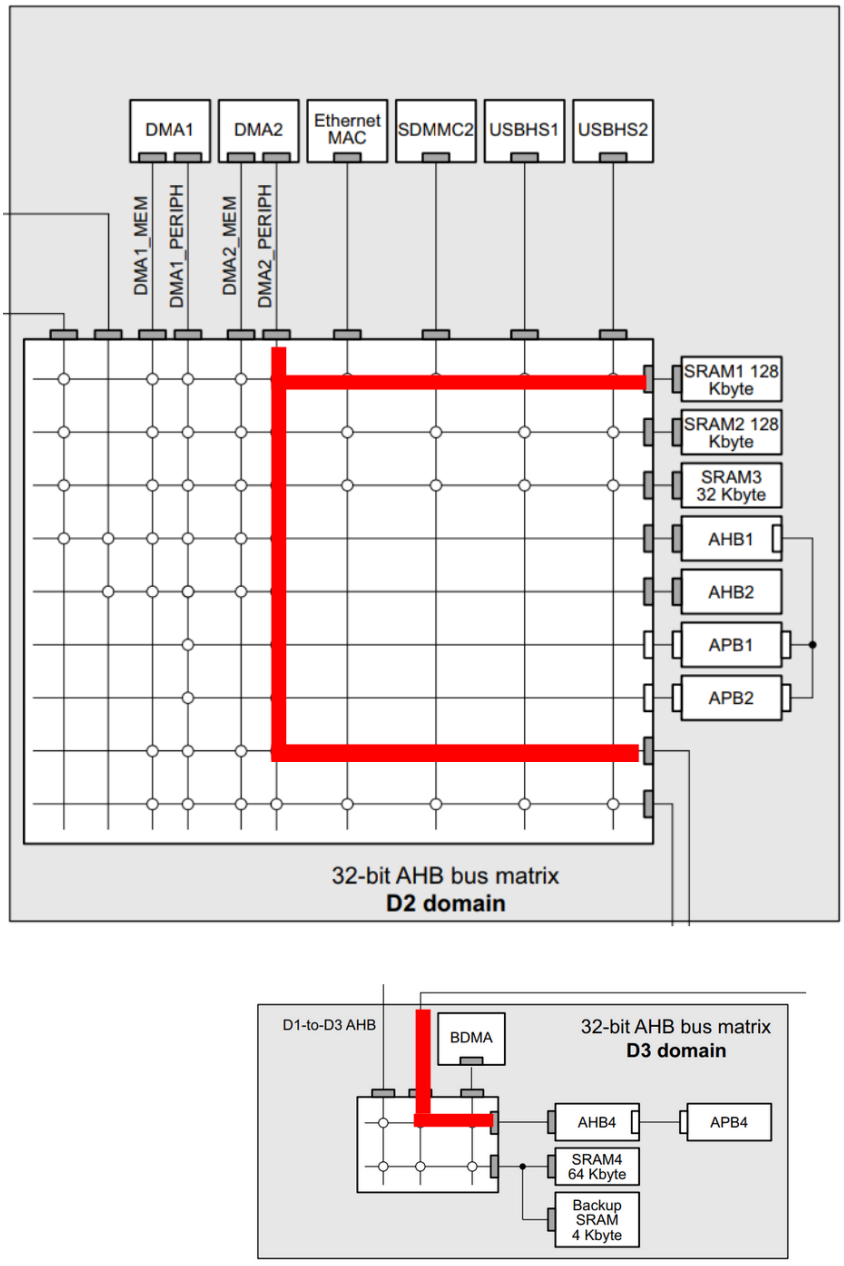


Figure 62: DMA Path from D2 Memory to GPIO

If for writing, The DMA is set to use a first-in, first-out (FIFO) buffer so that data can be queued to be written to GPIO. When there is no more data to send, the DMA stream will automatically

be disabled. A FIFO buffer is simpler to implement and requires less checks, but requires the knowledge of how much data to expect.

When reading data, a circular buffer is set because the amount of data the I/O Master needs to read is unknown. The DMA controller will continue to cycle and read data and it will be up to the firmware to manually disable the controller. The circular buffer will have the data shifted out of it every clock cycle until there is enough for a full frame. Afterwards, the frame will be put into an input queue and processed.

For both reading and writing data, Memory and Peripheral bursts can be configured to send more than a single chunk of data at one time. This is not desirable for keeping a consistent frequency when outputting data, so single burst mode is selected for all cases. The DMA must also be linked to the corresponding timer channel. This linking process will configure the data route so the clock signal is forwarded to the DMA controller as a trigger. The paired timer can use the external clock signal from the VCO or from the target device to trigger DMA. (CS)

5.3.10 USB Data Buffer

The USB data buffer will work with the built in STM32 HAL library functions. The following code initializes the USB device in USB 1.1 as a CDC device, which has been used for initial testing and configuration.

```
   /* Init Device Library, add supported class and start the library. */
   if (USBD_Init(&hUsbDeviceFS, &FS_Desc, DEVICE_FS) != USBD_OK) {
      Error_Handler();
   }
   if (USBD_RegisterClass(&hUsbDeviceFS, &USBD_CDC) != USBD_OK) {
      Error_Handler();
   }
   if (USBD_CDC_RegisterInterface(&hUsbDeviceFS, &USBD_Interface_fops_FS) !=
USBD_OK) {
```

```
    Error_Handler();
  }
  if (USBD_Start(&hUsbDeviceFS) != USBD_OK) {
    Error_Handler();
  }
  HAL_PWREx_EnableUSBVoltageDetector();
```

The USB hardware is initialized and the device is registered under the CDC class. The CDC device is then tied to the Full Speed USB interface and USB is enabled. Finally, the built in USB voltage detector is enabled to allow for hot plugging. To configure the USB hardware to work with USB 2.0 High Speed NCM, the following steps will be done:

- Initialize the USB device as a High Speed Device
- Register the USB device class as a CDC NCM class
- Register the device class to the USB High Speed interface
- Enable USB

In order to use USB 2.0 High Speed, an additional USB PHY chip is needed outside of the microcontroller. Without the chip, only USB 1.1 can be used, which is sufficient for lower speed communications used for demonstrations before the final board is completed.

To send data, the *CDC_Transmit_FS* function is used. This function takes in an input of a data buffer and the length of the buffer and will send the data over USB. If the transfer process is successful, it returns the *USBD_OK* error handle.

```
uint8_t CDC_Transmit_FS(uint8_t* Buf, uint16_t Len)
{
  uint8_t result = USBD_OK;
  /* USER CODE BEGIN 7 */
  USBD_CDC_HandleTypeDef *hcdc = (USBD_CDC_HandleTypeDef*)hUsbDeviceFS.pClassData;
  if (hcdc->TxState != 0){
```

```
      return USBD_BUSY;
    }
    USBD_CDC_SetTxBuffer(&hUsbDeviceFS, Buf, Len);
    result = USBD_CDC_TransmitPacket(&hUsbDeviceFS);
    /* USER CODE END 7 */
    return result;
  }
```

When the USB hardware receives data, an interrupt is triggered that calls the *CDC_Receive_FS* function. This function can set the flag notifying that there is new data to receive and process. To move the data out of the USB buffer, the following code is used:

```
uint8_t* Buf = malloc(USB_PACKET_SIZE);
USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
USBD_CDC_ReceivePakcet(&hUsbDeviceFS);
```

After the packet is processed, the buffer must be deleted using the *free* command. (CS/AD)

5.3.11 GUI and Device Driver

Figure 63 shows a level 1 block diagram of the GUI software. Table 56 lists the functional requirements of the block shown in Figure 63.

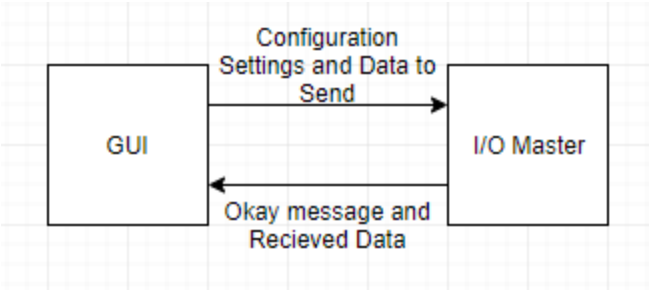


Figure 63: GUI Level 1 Block Diagram

Table 56: GUI Functional Requirements

Module	GUI
Designer	Corey Dye
Inputs	User input, Board input
Outputs	User output
Description	The GUI will send commands via USB that will configure the I/O Master to be able to accept the selected protocol. The User will be able to select a protocol from a drop down menu and the GUI will configure itself to prompt for whatever other inputs the selected protocol needs. The user will also be able to read inputs from the board in a text window as well as select items to send to the board. The GUI will be written in Python.

The interactive graphical user interface for this project is written in Python or more specifically, tkinter. The purpose for this part of the project is to make interacting with the I/O Master as intuitive yet customisable as possible. This means that the user should be able to set up the board as easily as possible yet they should still be able to customize the setup to suit their needs and be able to send some personalized commands to the board. The GUI will also send and receive data to and from the IO Master board so that the board may be configured appropriately for the users needs and data can be sent to the IO device with ease.

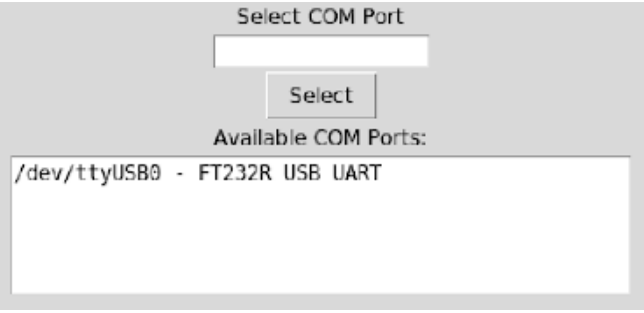


Figure 64: Serial Port Selection

The GUI starts out with a prompt to select an open COM port on the PC. This pop up box shows a list of all available COM ports on the PC and gives a brief description of what is connected to

that port as seen in Figure 64. This allows the user to easily figure out which port the IO Master is plugged into so that they can put the correct port into the input box and hit the select button. The program will then attempt to connect to that port. If the connect fails, the user is notified and asked to select another port. If the attempt is successful then the main dialog box for the program appears and the popup goes away. The user is then presented with a drop down that gives different options for protocols. After a protocol is selected, all of the needed entry fields will appear on the screen for the user to fill in. Some of these are text boxes that check user input to see if it makes sense for the board and also for the input field. Any selections are also displayed in the output text box so the user knows what data was sent to the board for configuration. This can all be seen in Figure 65. The user can change what protocol they are using at any time by changing the drop down at the top. There is also a custom packet command that sends hand built packets to allow for testing of specific commands and just to see if communication is working in general.

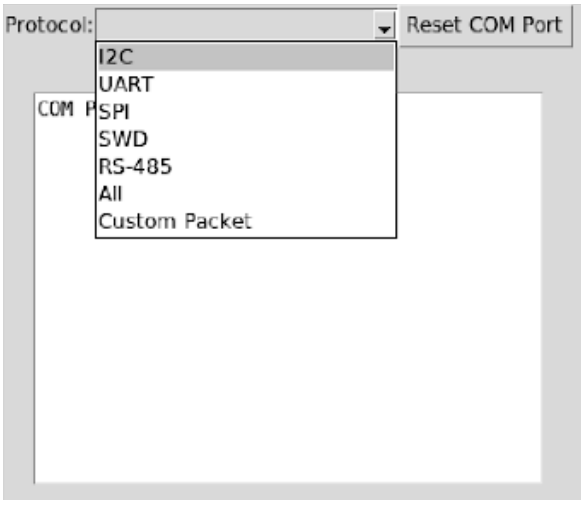


Figure 65: Protocol Selection Options

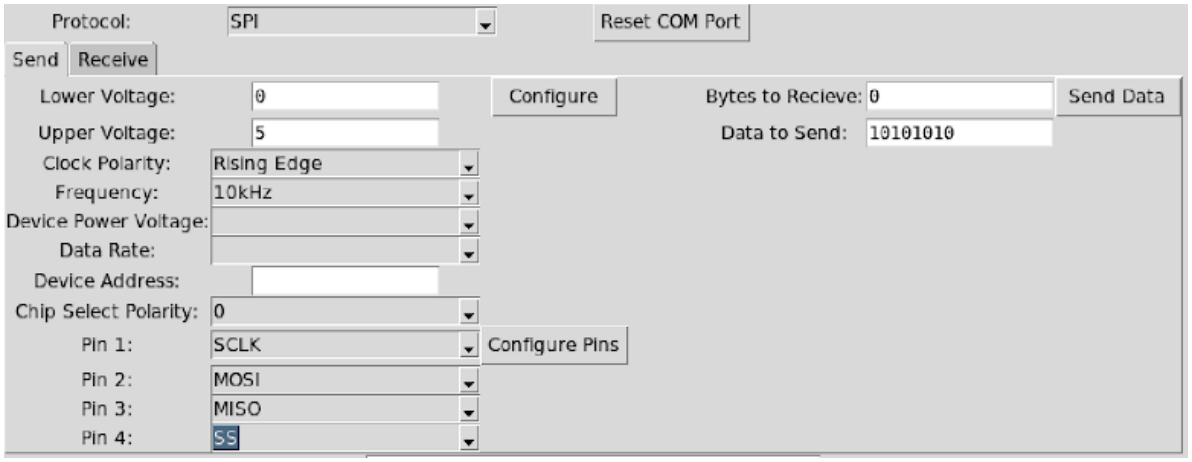


Figure 66: SPI Protocol Options

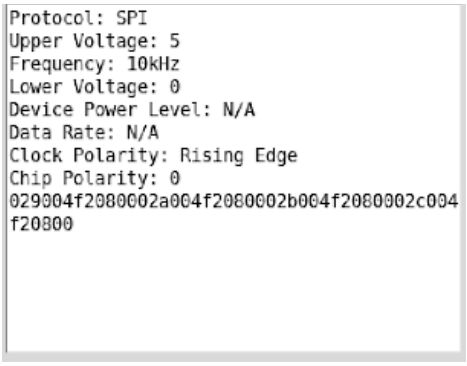


Figure 67: I/O Master Hardware Configuration Output

The code starts by declaring all the different packages that are needed from the Python library. It then goes into declaring variables for the window. This defines the size of the window and also what tabs are on the window. The first function defined is the initialize popup function. This function starts out by removing the main window from view and inserting a new popup window with two defined textboxes and a button. When the select button is pushed, a nested function named EntryCheck is called. This takes the string from the input textbox and tries to open a port with that name on the PC. If it works, then the main window returns to view and the popup goes

away. If the port fails to open then an error message appears and the user returns to the initializationPopup and tries again. This function also searches the computer for available COM ports and lists them in the output textbox. This allows the user to make an appropriate entry for the input box.

The next function in line order is read_from_port. This function was intended to be running in the background while it looks for data from the board to display to the user. Due to the lack of time to finish the project, this was never implemented properly.

Next is the sendCommand function. This function is fairly straightforward. It takes whatever is passed into the function and tries to send it using a serial write command. It either does this successfully or a message pop ups explaining that the port was unable to be written to. The initialization popup then appears so that a new COM port can be selected.

The next two functions are extended class functions for the Label class and the combobox class. These change the name of the built in functions and allows for easier assignments of these widgets. The input fields are formatted to properly create the element. While these may be slightly unnecessary now, earlier in the project they had more functionality and are now harder to take out than keep in.

ConfigureResults is what creates the packet to be sent before sending it to the SendCommand function. This function starts by getting all the inputs from all the available widgets then has different options depending on the protocol selected. If Custom Packet is selected, it will format all the input bytes and send them to the byteSendConfig function. After that, all the variables are shown to the user so that the user knows all the inputs that were sent to the board. This allows for

verification of correct data. Then, the voltage variables are checked for accuracy. If they are outside the range of the board (±15V) then an error message is given. Finally byteSendVolage is used to send the voltages to the board to configure all four pins.

setStateToConfig is used to configure the board so that it knows to be listening for configuration instructions. A short byte array is sent consisting of two 1s indicating a configure state to the board. A message is also given to the user stating that this happened.

byteSendVoltage is used to send voltage configurations to the board. A proper read/write bit is ORed with the appropriate command bits and resisterstate information to create byte 1 of the byte array. Then a math equation is used to get the voltage to a number that the board can work with easily and is then converted into 16 bits. The high and the low voltage are broken into 8 bit chunks then appended to the byte array. The byte array should then be 5 bytes long. Along with these another byte is used to tell the board what the rest of the bytes represent. So the 6 byte array is sent to the SendCommand function to be sent to the board. The sent packet is displayed to the user as well.

byteSendConfig will take in the read/write bit, the command bit, the resistor state and the following 4 bytes and convert them to a byte array. That array is then displayed to the user and then sent to the board using the SendCommand function. Any bytes that are not used should be inputted to the function using NULL.

setStateToLoadData is very similar to the setStateToConfig function except this function tells the board to load. A byte array consisting of 1 and 2 is created and sent to the board. A message is

then displayed to the output box for the user to see and SendCommand is called to send the array to the board.

Data_To_Send creates a byte array that sends user entered data to the board. An array consisting of a 3, 1, byteArrayLength, and the byte array is created. This tells the board that custom data is being sent to the IO device plugged into the IO Master and it states how long the data is so the board knows how long to keep searching for it. This function was not fully tested due to time constraints.

pinConfiguration‘s purpose is to set the 4 pins on the IO Master to either MOSI, SS, MISO, SCLK, or disable the pin. The function is currently only set up for SPI due to time constraints the other protocol options were never built into this function. After determining the protocol used, the function will set the four pins to the required type for that protocol. A future version would also display to the user the four pins and what they are set to for easy wiring of the device. The proper communication function is then created to send to the board. This function was never fully implemented due to lack of time.

receiveOrSend was created to allow for reading of data from the board so that it can be displayed to the user. This was never fully realized and the function is only half complete. The idea is to set the board to load state and then tell the board that the GUI expects to see so many bytes sent from the board. The GUI would then read in these bytes and show the user. Again, this was never tested and never fully coded.

The button class overwrites the built in button class the same way that label and combobox did earlier. The difference this time is that the button class also directs the code to other functions

when a button is pushed. When any button is pushed, it looks through the if statements in this class to figure out which button was pushed then the proper function is called.

The next several hundred lines of code all serve a similar purpose, to put widgets on the window. It starts out by creating all the labels, textboxes, comboboxes, buttons, etc and initializes them to say or do whatever is needed of them. This is followed by the display_create function. This function either places widgets on the screen or removes them from the screen. This is done by figuring out what protocol is in place then going through every widget and placing it on the grid or removing it from the grid based on whether or not the protocol needs that setting or not. This way only the most important and relevant items are on the screen at any time. This makes it easy for the user to see what is needed and what isn't for each protocol. An option is also in place to see all the widgets so that custom protocols can be used. The function is also bound to the protocol combobox. This means that whenever the protocol box is changed, the window updates with whatever is needed for the selected option.

The main function calls initialize popup and the mainloop. This lets the program start flowing appropriately. There is also commented out code that sets up multithreading. This was intended to let the program read data. This code worked early on for the end of semester demo in the fall but was never fully updated to allow for more complex receiving. Some of the code is in place for this to happen but due to the sudden end of the project, was never fully coded and therefore never tested.

This code was run and tested constantly throughout the process. Most of the time testing was possible by running the code dry on a computer by connecting to a random available COM port

and sending it data it couldn't read. This allowed testing for all of the send functions. The code was also tested using the prototype design board in the weeks leading up to the midterm demo. The data that the GUI was sending worked correctly as intending. This was when receiving was beginning to be implemented because testing receiving was difficult without any data to receive. This process was cut short due to the unfortunate and unforeseen circumstances. (CD)

```
import tkinter as TK
import usb.core
import usb.util
import serial
import time
import threading
import serial.tools.list_ports
import sys
#from tkinter import *
from tkinter import ttk
from tkinter import messagebox

window = TK.Tk()
window.title("IO Master Setup")
window.geometry("750x600")

tabControl = ttk.Notebook(window)
sendTab = ttk.Frame(tabControl)
receiveTab = ttk.Frame(tabControl)
tabControl.add(sendTab, text= 'Send')
tabControl.add(receiveTab, text='Receive')
tabControl.grid(column=0, row=1, columnspan=7) #expand=1, fill="both")

ser = serial.Serial()

def InitializePopup():
    window.withdraw()

    popup = TK.Toplevel()
    popup.title("Select Device Port")
    popup.geometry("450x185")

    txtlbl = TK.Label(popup, text = "Select COM Port")
    txtlbl.pack()

    txtin = TK.Text(popup, height=1, width=17)
    txtin.pack()

    EnterBtn = TK.Button(popup, text = "Select", command = lambda : EntryCheck())
    EnterBtn.pack()
```

```
    avlbl = TK.Label(popup, text = "Available COM Ports:")
    avlbl.pack()

    txtout = TK.Text(popup, height=5, width=50)
    txtout.pack()

    ports = list(serial.tools.list_ports.comports())
    for p in ports:
        txtout.insert(TK.END, p)
        txtout.insert(TK.END, '\n')
    txtout.configure(state = 'disable')

    def EntryCheck():
    #entry check for com port
        global ser
        strg = txtin.get('1.0', TK.END)
        try:
            ser = serial.Serial(strg.rstrip('\n'), 115200,  serial.EIGHTBITS,
serial.PARITY_NONE, serial.STOPBITS_ONE)
        except:
            messagebox.showinfo("Error", "Could not open COM port")
            #InitializePopup()
            return

        OutConfigTxt.configure(state="normal")
        OutConfigTxt.insert(TK.END, "COM Port: " + strg + '\n')
        OutConfigTxt.configure(state="disable")
        window.deiconify()
        popup.destroy()

def read_from_port(ser):
    while True:
        reading = ser.read()

        """
        print(reading)
        if reading == b'3':
            messagebox.showinfo("Works", "Pushed Button")
        """

def sendCommand(command):
    print(command.hex())
    try:
        ser.write(command)
    except:
        messagebox.showinfo("Error", "Could not write to COM port, verify the
device is plugged in then restate the correct COM port")
        InitializePopup()

class Label:
    def __init__(self, win, text):
        self.lbl=ttk.Label(win, text=text)
        #self.lbl.grid(column=clmn, row=row)
```

```
class combobox:
    def __init__(self, win, values):
        self.cb=ttk.Combobox(win, values=values, state = "readonly")

def Configure_Results():
    #Store VariablesI
    Protocol = cb0.cb.get()
    UVoltage = VHin.get('1.0', TK.END)
    Frequency = cb2.cb.get()
    LVoltage = VLin.get('1.0', TK.END)
    DPower = cb5.cb.get()
    DataRate = cb6.cb.get()
    ClPolarity = cb7.cb.get()
    ChPolarity = cb9.cb.get()

    OutConfigTxt.configure(state="normal")
    OutConfigTxt.delete('1.0', TK.END)

    if Protocol == "Custom Packet":
        rewr = int(WrRetxt.get('1.0', TK.END))
        commandbits = int(ComBittxt.get('1.0', TK.END))
        resisterStates = int(ResisStattxt.get('1.0', TK.END))

        byte2 = byte2in.get('1.0', TK.END)
        byte3 = byte3in.get('1.0', TK.END)
        byte4 = byte4in.get('1.0', TK.END)
        byte5 = byte5in.get('1.0', TK.END)

        if byte2.rstrip('\n') != 'NULL':
            byte2 = int(byte2)
        else:
            byte2 = byte2.rstrip('\n')

        if byte3.rstrip('\n') != 'NULL':
            byte3 = int(byte3)
        else:
            byte3 = byte3.rstrip('\n')

        if byte4.rstrip('\n') != 'NULL':
            byte4 = int(byte4)
        else:
            byte4 = byte4.rstrip('\n')

        if byte5.rstrip('\n') != 'NULL':
            byte5 = int(byte5)
        else:
            byte5 = byte5.rstrip('\n')

        byteSendConfig(rewr, commandbits, resisterStates, byte2, byte3, byte4,
byte5)
        return
```

```
    #if Protocol == "I2C":
    #if Protocol == "UART":
    #if Protocol == "SPI":
    #if Protocol == "SWD":
    #if Protocol == "RS-485":

    if Protocol == "":
        OutConfigTxt.insert(TK.END, "Protocol: N/A" + '\n')
    else:
        OutConfigTxt.insert(TK.END, "Protocol: " + Protocol + '\n')

    if UVoltage == "":
        OutConfigTxt.insert(TK.END, "Upper Voltage: N/A" + '\n')
    else:
        OutConfigTxt.insert(TK.END, "Upper Voltage: " + UVoltage)

    if Frequency == "":
        OutConfigTxt.insert(TK.END, "Frequency: N/A" + '\n')
    else:
        OutConfigTxt.insert(TK.END, "Frequency: " + Frequency + '\n')

    if LVoltage == "":
        OutConfigTxt.insert(TK.END, "Lower Voltage: N/A" + '\n')
    else:
        OutConfigTxt.insert(TK.END, "Lower Voltage: " + LVoltage)

    if DPower == "":
        OutConfigTxt.insert(TK.END, "Device Power Level: N/A" + '\n')
    else:
        OutConfigTxt.insert(TK.END, "Device Power Level: " + DPower + '\n')

    if DataRate == "":
        OutConfigTxt.insert(TK.END, "Data Rate: N/A" + '\n')
    else:
        OutConfigTxt.insert(TK.END, "Data Rate: " + DataRate + '\n')

    if ClPolarity == "":
        OutConfigTxt.insert(TK.END, "Clock Polarity: N/A" + '\n')
    else:
        OutConfigTxt.insert(TK.END, "Clock Polarity: " + ClPolarity + '\n')

    if ChPolarity == "":
        OutConfigTxt.insert(TK.END, "Chip Polarity: N/A" + '\n')
    else:
        OutConfigTxt.insert(TK.END, "Chip Polarity: " + ChPolarity + '\n')

    OutConfigTxt.configure(state="disable")

    if (float(UVoltage) > 15 or float(UVoltage) < float(LVoltage)):
        #print(float(UVoltage))
        #print(float(LVoltage))
        messagebox.showinfo("Error", "Invalid Data.\n Pick a voltage of 15V or
Lower. \n Also, Upper Voltage should be larger than Lower Voltage")
```

```
        return
    else:
        VH = float(UVoltage)

    if (float(LVoltage) < -15 or float(UVoltage) < float(LVoltage)):
        messagebox.showinfo("Error", "Invalid Data.\n Pick a voltage of -15V or
Higher for Lower Voltage. \n Also, Upper Voltage should be larger than Lower
Voltage")
        return
    else:
        VL = float(LVoltage)

    #rewr=1 because we are writing, commandbits determines what we are setting,
resister State is unknown
    #rewr = int(WrRetxt.get('1.0', TK.END))
    #resisterStates = int(ResisStattxt.get('1.0', TK.END))

    byteSendVoltage(1, 1, 0, VH, VL)
    byteSendVoltage(1, 2, 0, VH, VL)
    byteSendVoltage(1, 3, 0, VH, VL)
    byteSendVoltage(1, 4, 0, VH, VL)


    #End Function

def setStateToConfig():
    #send a 2 byte array of 1 and 1
    Config_Bytes = bytearray()

    Config_Bytes.append(1)
    Config_Bytes.append(1)
    sendCommand(Config_Bytes)

    OutConfigTxt.configure(state="normal")
    OutConfigTxt.insert(TK.END, "State Set to Configure: " + Config_Bytes.hex() +
'\n')
    OutConfigTxt.configure(state="disable")


def byteSendVoltage(rewr, commandbits, resisterStates, VH, VL):
    VH = VH + 1.3
    #rewr=1 bit, commandbits=3, resisterstate=4 bits
    rewr = rewr << 7
    combits = commandbits << 4

    commandByte = rewr | combits | resisterStates #byte 1

    TVH = round(((2**12 - 1) / 33) * (16.5 - VH))
    TVL = round(((2**12 - 1) / 33) * (16.5 - VL))
    print("TVH")
    print(TVH)
    print("TVL")
    print(TVL)
```

```
    UVH = (TVH >> 8) & 0xff
    LVH = (TVH) & 0xff
    UVL = (TVL >> 8) & 0xff
    LVL = (TVL) & 0xff

    OutConfigTxt.configure(state="normal")

    Packet_Bytes = bytearray()
    Packet_Bytes.append(2) #Byte0
    Packet_Bytes.append(commandByte)#byte1
    Packet_Bytes.append(UVH)#byte2
    Packet_Bytes.append(LVH)#byte3
    Packet_Bytes.append(UVL)#byte4
    Packet_Bytes.append(LVL)#byte5

    OutConfigTxt.insert(TK.END, Packet_Bytes.hex())
    OutConfigTxt.configure(state="disable")

    sendCommand(Packet_Bytes)
    time.sleep(.1)


#Setting any of the 4 bytes to NULL will skip that byte in the array
def byteSendConfig(rewr, commandbits, resisterStates, byte2, byte3, byte4, byte5):
    #set state of the board to configure
    setStateToConfig()

    #rewr=1 bit, commandbits=3, resisterstate=4 bits
    rewr = rewr << 7
    combits = commandbits << 4

    commandByte = rewr | combits | resisterStates #byte 1

    OutConfigTxt.configure(state="normal")

    Packet_Bytes = bytearray()

    Packet_Bytes.append(2) #Config Command Byte

    Packet_Bytes.append(commandByte)#byte1
    if byte2 != 'NULL':
        Packet_Bytes.append(byte2)#byte2
    if byte3 != 'NULL':
        Packet_Bytes.append(byte3)#byte3
    if byte4 != 'NULL':
        Packet_Bytes.append(byte4)#byte4
    if byte5 != 'NULL':
        Packet_Bytes.append(byte5)#byte5

    OutConfigTxt.insert(TK.END, "Data Sent: " + Packet_Bytes.hex() + '\n')
    OutConfigTxt.configure(state="disable")
```

```
    sendCommand(Packet_Bytes)

def setStateToLoadData():
    #send a 2 byte array of 1 and 1
    Config_Bytes = bytearray()

    Config_Bytes.append(1)
    Config_Bytes.append(2)
    sendCommand(Config_Bytes)

    OutConfigTxt.configure(state="normal")
    OutConfigTxt.insert(TK.END, "State Set to Configure: " + Config_Bytes.hex() +
'\n')
    OutConfigTxt.configure(state="disable")

def Data_To_Send():
    #convert text to byte array.
    Data_Bytes = bytearray()
    dataToSend = DataText3.get('1.0', TK.END)
    for element in dataToSend.rstrip('\n'):
        Data_Bytes.append(ord(element))


    #Make packet to send
    byteArrayLength = len(Data_Bytes)
    Data_Send_Bytes = bytearray()
    Data_Send_Bytes.append(3)
    Data_Send_Bytes.append(1)
    Data_Send_Bytes.append(byteArrayLength)
    Data_Send_Bytes.extend(Data_Bytes)

    sendCommand(Data_Send_Bytes)

    OutConfigTxt.configure(state="normal")
    OutConfigTxt.insert(TK.END, "Data Sent: " + Data_Send_Bytes.hex() + '\n')
    OutConfigTxt.configure(state="disable")

def pinConfiguration():
    #Determine bits
    def getBitConfig(strEntry):
        if strEntry == "MOSI" or strEntry == "SS":
            return 2    #Outpin
        elif strEntry == "MISO":
            return 1    #InPin
        elif strEntry == "SCLK":
            return 3    #ClockPin
        else:
            return 0    #pin Disabled

    #set Pin Configuration for SPI based on user input
    if cb0.cb.get() == "SPI":
        pin1 = pin1spi.cb.get()
        pin2 = pin2spi.cb.get()
```

```
        pin3 = pin3spi.cb.get()
        pin4 = pin4spi.cb.get()

        if pin1 == pin2 or pin1 == pin3 or pin1 == pin4 or pin2 == pin3 or pin2 ==
pin4 or pin3 == pin4:
            messagebox.showinfo("Error", "Invalid Data.\n The pins should all be
unique")
            return

    Abits = getBitConfig(pin1)
    Bbits = getBitConfig(pin2)
    Cbits = getBitConfig(pin3)
    Dbits = getBitConfig(pin4)

    #byte 1
    rewr = 1
    commandbits = 7
    resisterStates = 0

    #byte 2
    pinPattern = (Dbits << 6) | (Cbits << 4) | (Bbits << 2) | (Abits)

    byteSendConfig(rewr,commandbits,resisterStates,pinPattern,'NULL','NULL','NULL')

def receiveOrSend():
    try:
        bytes2find = int(B2Rtxt.get('1.0', TK.END))
    except:
        messagebox.showinfo("Error", "Enter a Valid Positive Number for Bytes to
Read")
        return
    if bytes2find < 0:
        messagebox.showinfo("Error", "Enter a Valid Positive Number for Bytes to
Read")
        return

    # Send designated Load Data Mode
    setStateToLoadData()

    #Check how many bytes to recieve before sending data
    if bytes2find == 0:
        Data_To_Send()
        return

    #send Bytes to read command
    B2S_Bytes = bytearray()
    B2S_Bytes.append(2)
    B2S_Bytes.append(bytes2find)
    sendCommand(B2S_Bytes)

    #Else Look for the amount of Bytes entered
    receivedBytes = bytearray()
    #read in Byte
```

```
    #receivedBytes = ser.read(bytes2find)    #line causes code to stop responding.
Cause unknown

    OutConfigTxt.configure(state="normal")
    OutConfigTxt.insert(TK.END, "Looked for Byte: " + receivedBytes.hex() + '\n')
    OutConfigTxt.configure(state="disable")

class Button:
    def __init__(self, win, text):
        self.btn=ttk.Button(win, text=text, command=self.press)

    def press(self):
        btn_text = self.btn.cget('text')
        if btn_text == "Configure":
            setStateToConfig()
            Configure_Results()

        if btn_text == "Send Data":
            receiveOrSend()

        if btn_text == "Reset COM Port":
            InitializePopup()

        if btn_text == "Configure Pins":
            pinConfiguration()

#Each object needs to be created ouside the function and placed on the window in
the function

lbl0 = Label(window, "Protocol:")    #Protocol
lbl1 = Label(sendTab, "Upper Voltage:")    #Upper Voltage
lbl2 = Label(sendTab, "Frequency:")    #Frequency
Data2Send = Label(sendTab, "Data to Send:")    #Data to Send
lbl4 = Label(sendTab, "Lower Voltage:")    #Lower Voltage
lbl5 = Label(sendTab, "Device Power Voltage:")    #Device Power Voltage
lbl6 = Label(sendTab, "Data Rate:")    #Data Rate
lbl7 = Label(sendTab, "Clock Polarity:")    #Clock Polarity (Rise or fall)
lbl8 = Label(sendTab, "Device Address:")    #Device Address
lbl9 = Label(sendTab, "Chip Select Polarity:")    #Chip Select Polarity
WrRelbl = Label(sendTab, "Read Write bit:")    #read write bit for custom packet
send (1 bit)
ComBitlbl = Label(sendTab, "Commandbits:")    #Set command bits (3 bits)
ResisStatlbl = Label(sendTab, "Resister State:")    #set resister state (either 0,
1, 2)(4 bits)

pin1lbl = Label(sendTab, "Pin 1:")    #Create Label for Pin1 dropdown
pin2lbl = Label(sendTab, "Pin 2:")    #Create Label for Pin2 dropdown
pin3lbl = Label(sendTab, "Pin 3:")    #Create Label for Pin3 dropdown
pin4lbl = Label(sendTab, "Pin 4:")    #Create Label for Pin4 dropdown

lbl0.lbl.grid(column=0, row=0)    #place protocol selection label (Always On)
```

```
cb0 = combobox(window, ["I2C", "UART", "SPI", "SWD", "RS-485", "All", "Custom
Packet"])    #create drop down for protocol selection
VHin = TK.Text(sendTab, height=1, width=17)    #Voltage Selection
cb2 = combobox(sendTab, ["1kHz", "10kHz", "100kHz", "1MHz"])    #Frequency
Selection
VLin = TK.Text(sendTab, height=1, width=17)    #Lower volatage level
cb5 = combobox(sendTab, ["3.3V","5V", "12V", "24V" ])    #Device Power Level
cb6 = combobox(sendTab, ["Data Rates"])    #Data Rates
cb7 = combobox(sendTab, ["Rising Edge", "Falling edge"])    #Clock Polarity
cb9 = combobox(sendTab, ["0", "1"])    #Chip Select Polarity

pin1spi = combobox(sendTab, ["MOSI", "MISO", "SCLK", "SS"])    #Combobox options
for Pin for SPI
pin2spi = combobox(sendTab, ["MOSI", "MISO", "SCLK", "SS"])    #Combobox options
for Pin for SPI
pin3spi = combobox(sendTab, ["MOSI", "MISO", "SCLK", "SS"])    #Combobox options
for Pin for SPI
pin4spi = combobox(sendTab, ["MOSI", "MISO", "SCLK", "SS"])    #Combobox options
for Pin for SPI


cb0.cb.grid(column=1, row=0)    #Place drop down for protocols

DataText3 = TK.Text(sendTab, height=1, width=17)    #Box to enter 8 bit command to
board (gets checked)
AddressText = TK.Text(sendTab, height=1, width=17)    #Box to enter the device
address.
OutConfigTxt = TK.Text(window, height = 15, width = 42, state = "disabled")
#Display sent configurables in this box
OutConfigTxt.grid(column=0, row=6, columnspan=7, padx=(0,0))

WrRetxt = TK.Text(sendTab, height=1, width=17)    #read write bit for custom packet
send (1 bit)
WrRetxt.insert(TK.END, "1")
ComBittxt = TK.Text(sendTab, height=1, width=17)    #Set command bits (3 bits)
ComBittxt.insert(TK.END, "7")
ResisStattxt = TK.Text(sendTab, height=1, width=17)    #set resister state (either
0, 1, 2)(4 bits)
ResisStattxt.insert(TK.END, "0")


btn1 = Button(sendTab, "Configure")    #Send configure
Data2SendBtn = Button(sendTab, "Send Data")
ReComBtn = Button(window, "Reset COM Port")
ReComBtn.btn.grid(column = 2, row = 0)
pinSetBtn = Button(sendTab, "Configure Pins")

#Byte Input Config
b2lbl = Label(sendTab, "Byte 2 (As Integer or 'NULL'): ")    #Input for Byte 2
Label
b3lbl = Label(sendTab, "Byte 3 (As Integer or 'NULL'): ")    #Input for Byte 3
Label
```

```
b4lbl = Label(sendTab, "Byte 4 (As Integer or 'NULL'): ")    #Input for Byte 4
Label
b5lbl = Label(sendTab, "Byte 5 (As Integer or 'NULL'): ")    #Input for Byte 5
Label
byte2in = TK.Text(sendTab, height=1, width=17)    #input for manual packet
composition for Byte 2
byte3in = TK.Text(sendTab, height=1, width=17)    #input for manual packet
composition for Byte 3
byte4in = TK.Text(sendTab, height=1, width=17)    #input for manual packet
composition for Byte 4
byte5in = TK.Text(sendTab, height=1, width=17)    #input for manual packet
composition for Byte 5

#SendReceieve textbox
B2Rlbl = Label(sendTab, "Bytes to Recieve:")    #Bytes to Receive Label
B2Rtxt = TK.Text(sendTab, height=1, width=17)    #Bytes to Receive Input
B2Rtxt.insert(TK.END, "0")    #Preset to 0

#Choose which objects are displayed based on the protocol chosen

def display_create(window):
    #Create Interface for SPI
    if cb0.cb.get() == 'SPI':
        lbl1.lbl.grid(column=0, row=3)    #Upper Voltage
        lbl2.lbl.grid(column=0, row=5)    #Frequency
        lbl4.lbl.grid(column=0, row=2)    #Lower Voltage
        lbl5.lbl.grid(column=0, row=6)    #Device Power
        lbl6.lbl.grid(column=0, row=7)    #Data Rate
        lbl7.lbl.grid(column=0, row=4)    #Clock Polarity
        lbl8.lbl.grid(column=0, row=8)    #Device Address
        lbl9.lbl.grid(column=0, row=9)    #Chip Select Polarity

        WrRelbl.lbl.grid_forget()    #read write bit for custom packet send (1 bit)
        ComBitlbl.lbl.grid_forget()    #Set command bits (3 bits)
        ResisStatlbl.lbl.grid_forget()    #set resister state (either 0, 1, 2)(4
bits)

        VHin.grid(column=1, row=3)    #Upper Voltage
        cb2.cb.grid(column=1, row=5)    #Frequency
        VLin.grid(column=1, row=2)    #Lower Voltage Level
        cb5.cb.grid(column=1, row=6)    #Device Power
        cb6.cb.grid(column=1, row=7)    #Data Rates
        cb7.cb.grid(column=1, row=4)    #Clock Polarity
        cb9.cb.grid(column=1, row=9)    #Chip Select Polarity

        #Pin Settings
        pin1lbl.lbl.grid(column=0, row=11)    #Create Label for Pin1 dropdown
        pin2lbl.lbl.grid(column=0, row=12)    #Create Label for Pin2 dropdown
        pin3lbl.lbl.grid(column=0, row=13)    #Create Label for Pin3 dropdown
        pin4lbl.lbl.grid(column=0, row=14)    #Create Label for Pin4 dropdown
        pin1spi.cb.grid(column=1, row=11)    #Combobox options for Pin for SPI
        pin2spi.cb.grid(column=1, row=12)    #Combobox options for Pin for SPI
        pin3spi.cb.grid(column=1, row=13)    #Combobox options for Pin for SPI
```

```
        pin4spi.cb.grid(column=1, row=14)    #Combobox options for Pin for SPI
        pinSetBtn.btn.grid(column=2, row=11)    #Send the configuration of the pins
to the board

        DataText3.grid(column=1, row=10)    #Data to send
        AddressText.grid(column=1, row=8)    #Device Address Box

        WrRetxt.grid_forget()    #read write bit for custom packet send (1 bit)
        ComBittxt.grid_forget()    #Set command bits (3 bits)
        ResisStattxt.grid_forget()    #set resister state (either 0, 1, 2)(4 bits)

        btn1.btn.grid(column=2, row=2)    #Send configure

        #Byte Input Config
        b2lbl.lbl.grid_forget()    #Input for Byte 2 Label
        b3lbl.lbl.grid_forget()    #Input for Byte 3 Label
        b4lbl.lbl.grid_forget()    #Input for Byte 4 Label
        b5lbl.lbl.grid_forget()    #Input for Byte 5 Label
        byte2in.grid_forget()    #input for manual packet composition for Byte 2
        byte3in.grid_forget()    #input for manual packet composition for Byte 3
        byte4in.grid_forget()    #input for manual packet composition for Byte 4
        byte5in.grid_forget()    #input for manual packet composition for Byte 5

        #SendReceieve textbox
        B2Rlbl.lbl.grid(column=4, row=2, padx=(50,0))    #Bytes to Receive Label
        B2Rtxt.grid(column=5, row=2)    #Bytes to Receive Input
        Data2Send.lbl.grid(column=4, row=3, padx=(50,0))    #Data to send
        DataText3.grid(column=5, row=3)    #Data to send
        Data2SendBtn.btn.grid(column=6, row=2)    #Send Data from text box

    #Display I2C Components
    if cb0.cb.get() == 'I2C':
        lbl1.lbl.grid(column=0, row=3)    #Upper Voltage
        lbl2.lbl.grid(column=0, row=5)    #Frequency
        lbl4.lbl.grid(column=0, row=2)    #Lower Voltage
        lbl5.lbl.grid(column=0, row=6)    #Device Power
        lbl6.lbl.grid(column=0, row=7)    #Data Rate
        lbl7.lbl.grid(column=0, row=4)    #Clock Polarity
        lbl8.lbl.grid(column=0, row=8)    #Device Address
        lbl9.lbl.grid_forget()    #Chip Select Polarity

        WrRelbl.lbl.grid_forget()    #read write bit for custom packet send (1 bit)
        ComBitlbl.lbl.grid_forget()    #Set command bits (3 bits)
        ResisStatlbl.lbl.grid_forget()    #set resister state (either 0, 1, 2)(4
bits)

        VHin.grid(column=1, row=3)    #Upper Voltage
        cb2.cb.grid(column=1, row=5)    #Frequency
        VLin.grid(column=1, row=2)    #Lower Voltage Level
        cb5.cb.grid(column=1, row=6)    #Device Power
        cb6.cb.grid(column=1, row=7)    #Data Rates
        cb7.cb.grid(column=1, row=4)    #Clock Polarity
        cb9.cb.grid_forget()    #Chip Select Polarity
```

```
        #Empty Unused Boxes
        cb9.cb.set('')
        AddressText.grid(column=1, row=8)    #Device Address Box

        WrRetxt.grid_forget()    #read write bit for custom packet send (1 bit)
        ComBittxt.grid_forget()    #Set command bits (3 bits)
        ResisStattxt.grid_forget()    #set resister state (either 0, 1, 2)(4 bits)

        btn1.btn.grid(column=2, row=2)    #Send configure

        #Pin Labels
        pin1lbl.lbl.grid(column=0, row=11)    #Create Label for Pin1 dropdown
        pin2lbl.lbl.grid(column=0, row=12)    #Create Label for Pin2 dropdown
        pin3lbl.lbl.grid(column=0, row=13)    #Create Label for Pin3 dropdown
        pin4lbl.lbl.grid(column=0, row=14)    #Create Label for Pin4 dropdown
        pin1spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin2spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin3spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin4spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pinSetBtn.btn.grid(column=2, row=11)    #Send the configuration of the pins
to the board

        #Byte Input Config
        b2lbl.lbl.grid_forget()    #Input for Byte 2 Label
        b3lbl.lbl.grid_forget()    #Input for Byte 3 Label
        b4lbl.lbl.grid_forget()    #Input for Byte 4 Label
        b5lbl.lbl.grid_forget()    #Input for Byte 5 Label
        byte2in.grid_forget()    #input for manual packet composition for Byte 2
        byte3in.grid_forget()    #input for manual packet composition for Byte 3
        byte4in.grid_forget()    #input for manual packet composition for Byte 4
        byte5in.grid_forget()    #input for manual packet composition for Byte 5

        #SendReceieve textbox
        B2Rlbl.lbl.grid(column=4, row=2, padx=(50,0))    #Bytes to Receive Label
        B2Rtxt.grid(column=5, row=2)    #Bytes to Receive Input
        Data2Send.lbl.grid(column=4, row=3, padx=(50,0))    #Data to send
        DataText3.grid(column=5, row=3)    #Data to send
        Data2SendBtn.btn.grid(column=6, row=2)    #Send Data from text box


    #Display UART Components
    if cb0.cb.get() == 'UART':
        lbl1.lbl.grid(column=0, row=3)    #Upper Voltage
        lbl2.lbl.grid_forget()    #Frequency
        lbl4.lbl.grid(column=0, row=2)    #Lower Voltage
        lbl5.lbl.grid(column=0, row=6)    #Device Power
        lbl6.lbl.grid(column=0, row=7)    #Data Rate
        lbl7.lbl.grid(column=0, row=4)    #Clock Polarity
        lbl8.lbl.grid_forget()    #Device Address
        lbl9.lbl.grid_forget()    #Chip Select Polarity

        WrRelbl.lbl.grid_forget()    #read write bit for custom packet send (1 bit)
```

```
        ComBitlbl.lbl.grid_forget()    #Set command bits (3 bits)
        ResisStatlbl.lbl.grid_forget()    #set resister state (either 0, 1, 2)(4
bits)

        VHin.grid(column=1, row=3)    #Upper Voltage
        cb2.cb.grid_forget()    #Frequency
        VLin.grid(column=1, row=2)    #Lower Voltage Level
        cb5.cb.grid(column=1, row=6)    #Device Power
        cb6.cb.grid(column=1, row=7)    #Data Rates
        cb7.cb.grid(column=1, row=4)    #Clock Polarity
        cb9.cb.grid_forget()    #Chip Select Polarity

        #Empty Unused Boxes
        cb2.cb.set('')
        cb9.cb.set('')
        AddressText.delete('1.0', TK.END)

        AddressText.grid_forget()    #Device Address Box

        WrRetxt.grid_forget()    #read write bit for custom packet send (1 bit)
        ComBittxt.grid_forget()    #Set command bits (3 bits)
        ResisStattxt.grid_forget()    #set resister state (either 0, 1, 2)(4 bits)

        btn1.btn.grid(column=2, row=2)    #Send configure

        #Pin Labels
        pin1lbl.lbl.grid(column=0, row=11)    #Create Label for Pin1 dropdown
        pin2lbl.lbl.grid(column=0, row=12)    #Create Label for Pin2 dropdown
        pin3lbl.lbl.grid(column=0, row=13)    #Create Label for Pin3 dropdown
        pin4lbl.lbl.grid(column=0, row=14)    #Create Label for Pin4 dropdown
        pin1spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin2spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin3spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin4spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pinSetBtn.btn.grid(column=2, row=11)    #Send the configuration of the pins
to the board

        #Byte Input Config
        b2lbl.lbl.grid_forget()    #Input for Byte 2 Label
        b3lbl.lbl.grid_forget()    #Input for Byte 3 Label
        b4lbl.lbl.grid_forget()    #Input for Byte 4 Label
        b5lbl.lbl.grid_forget()    #Input for Byte 5 Label
        byte2in.grid_forget()    #input for manual packet composition for Byte 2
        byte3in.grid_forget()    #input for manual packet composition for Byte 3
        byte4in.grid_forget()    #input for manual packet composition for Byte 4
        byte5in.grid_forget()    #input for manual packet composition for Byte 5

        #SendReceieve textbox
        B2Rlbl.lbl.grid(column=4, row=2, padx=(50,0))    #Bytes to Receive Label
        B2Rtxt.grid(column=5, row=2)    #Bytes to Receive Input
        Data2Send.lbl.grid(column=4, row=3, padx=(50,0))    #Data to send
        DataText3.grid(column=5, row=3)    #Data to send
        Data2SendBtn.btn.grid(column=6, row=2)    #Send Data from text box
```

```
    #Display SWD Components
    if cb0.cb.get() == 'SWD':
        lbl1.lbl.grid(column=0, row=3)    #Upper Voltage
        lbl2.lbl.grid_forget()    #Frequency
        lbl4.lbl.grid(column=0, row=2)    #Lower Voltage
        lbl5.lbl.grid(column=0, row=6)    #Device Power
        lbl6.lbl.grid_forget()    #Data Rate
        lbl7.lbl.grid(column=0, row=4)    #Clock Polarity
        lbl8.lbl.grid_forget()    #Device Address
        lbl9.lbl.grid_forget()    #Chip Select Polarity

        WrRelbl.lbl.grid_forget()    #read write bit for custom packet send (1 bit)
        ComBitlbl.lbl.grid_forget()    #Set command bits (3 bits)
        ResisStatlbl.lbl.grid_forget()    #set resister state (either 0, 1, 2)(4
bits)

        VHin.grid(column=1, row=3)    #Upper Voltage
        cb2.cb.grid_forget()    #Frequency
        VLin.grid(column=1, row=2)    #Lower Voltage Level
        cb5.cb.grid(column=1, row=6)    #Device Power
        cb6.cb.grid_forget()    #Data Rates
        cb7.cb.grid(column=1, row=4)    #Clock Polarity
        cb9.cb.grid_forget()    #Chip Select Polarity

        #Empty Unused Boxes
        cb2.cb.set('')
        cb6.cb.set('')
        cb9.cb.set('')
        AddressText.delete('1.0', TK.END)
        DataText3.delete('1.0', TK.END)

        AddressText.grid_forget()    #Device Address Box

        WrRetxt.grid_forget()    #read write bit for custom packet send (1 bit)
        ComBittxt.grid_forget()    #Set command bits (3 bits)
        ResisStattxt.grid_forget()    #set resister state (either 0, 1, 2)(4 bits)

        btn1.btn.grid(column=2, row=2)    #Send configure


        #Pin Labels
        pin1lbl.lbl.grid(column=0, row=11)    #Create Label for Pin1 dropdown
        pin2lbl.lbl.grid(column=0, row=12)    #Create Label for Pin2 dropdown
        pin3lbl.lbl.grid(column=0, row=13)    #Create Label for Pin3 dropdown
        pin4lbl.lbl.grid(column=0, row=14)    #Create Label for Pin4 dropdown
        pin1spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin2spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin3spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin4spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pinSetBtn.btn.grid(column=2, row=11)    #Send the configuration of the pins
to the board
```

```
        #Byte Input Config
        b2lbl.lbl.grid_forget()    #Input for Byte 2 Label
        b3lbl.lbl.grid_forget()    #Input for Byte 3 Label
        b4lbl.lbl.grid_forget()    #Input for Byte 4 Label
        b5lbl.lbl.grid_forget()    #Input for Byte 5 Label
        byte2in.grid_forget()    #input for manual packet composition for Byte 2
        byte3in.grid_forget()    #input for manual packet composition for Byte 3
        byte4in.grid_forget()    #input for manual packet composition for Byte 4
        byte5in.grid_forget()    #input for manual packet composition for Byte 5

        #SendReceieve textbox
        B2Rlbl.lbl.grid_forget()    #Bytes to Receive Label
        B2Rtxt.grid_forget()    #Bytes to Receive Input
        Data2Send.lbl.grid_forget()    #Data to send
        DataText3.grid_forget()    #Data to send
        Data2SendBtn.btn.grid_forget()    #Send Data from text box


    #display RS-485 Components
    if cb0.cb.get() == 'RS-485':
        lbl1.lbl.grid(column=0, row=3)    #Upper Voltage
        lbl2.lbl.grid_forget()    #Frequency
        lbl4.lbl.grid(column=0, row=2)    #Lower Voltage
        lbl5.lbl.grid(column=0, row=6)    #Device Power
        lbl6.lbl.grid(column=0, row=7)    #Data Rate
        lbl7.lbl.grid_forget()    #Clock Polarity
        lbl8.lbl.grid_forget()    #Device Address
        lbl9.lbl.grid_forget()    #Chip Select Polarity

        WrRelbl.lbl.grid_forget()    #read write bit for custom packet send (1 bit)
        ComBitlbl.lbl.grid_forget()    #Set command bits (3 bits)
        ResisStatlbl.lbl.grid_forget()    #set resister state (either 0, 1, 2)(4
bits)

        VHin.grid(column=1, row=3)    #Upper Voltage
        cb2.cb.grid_forget()    #Frequency
        VLin.grid(column=1, row=2)    #Lower Voltage Level
        cb5.cb.grid(column=1, row=6)    #Device Power
        cb6.cb.grid(column=1, row=7)    #Data Rates
        cb7.cb.grid_forget()    #Clock Polarity
        cb9.cb.grid_forget()    #Chip Select Polarity

        #Empty Unused Boxes
        cb2.cb.set('')
        cb7.cb.set('')
        cb9.cb.set('')
        AddressText.delete('1.0', TK.END)

        AddressText.grid_forget()    #Device Address Box

        WrRetxt.grid_forget()    #read write bit for custom packet send (1 bit)
        ComBittxt.grid_forget()    #Set command bits (3 bits)
```

```
        ResisStattxt.grid_forget()    #set resister state (either 0, 1, 2)(4 bits)

        btn1.btn.grid(column=2, row=2)    #Send configure

        #Pin Labels
        pin1lbl.lbl.grid(column=0, row=11)    #Create Label for Pin1 dropdown
        pin2lbl.lbl.grid(column=0, row=12)    #Create Label for Pin2 dropdown
        pin3lbl.lbl.grid(column=0, row=13)    #Create Label for Pin3 dropdown
        pin4lbl.lbl.grid(column=0, row=14)    #Create Label for Pin4 dropdown
        pin1spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin2spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin3spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin4spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pinSetBtn.btn.grid(column=2, row=11)    #Send the configuration of the pins
to the board

        #Byte Input Config
        b2lbl.lbl.grid_forget()    #Input for Byte 2 Label
        b3lbl.lbl.grid_forget()    #Input for Byte 3 Label
        b4lbl.lbl.grid_forget()    #Input for Byte 4 Label
        b5lbl.lbl.grid_forget()    #Input for Byte 5 Label
        byte2in.grid_forget()    #input for manual packet composition for Byte 2
        byte3in.grid_forget()    #input for manual packet composition for Byte 3
        byte4in.grid_forget()    #input for manual packet composition for Byte 4
        byte5in.grid_forget()    #input for manual packet composition for Byte 5

        #SendReceieve textbox
        B2Rlbl.lbl.grid(column=4, row=2, padx=(50,0))    #Bytes to Receive Label
        B2Rtxt.grid(column=5, row=2)    #Bytes to Receive Input
        Data2Send.lbl.grid(column=4, row=3, padx=(50,0))    #Data to send
        DataText3.grid(column=5, row=3)    #Data to send
        Data2SendBtn.btn.grid(column=6, row=2)    #Send Data from text box

    #Display Custom Options
    if cb0.cb.get() == 'All':
        lbl1.lbl.grid(column=0, row=3)    #Upper Voltage
        lbl2.lbl.grid(column=0, row=5)    #Frequency
        lbl4.lbl.grid(column=0, row=2)    #Lower Voltage
        lbl5.lbl.grid(column=0, row=6)    #Device Power
        lbl6.lbl.grid(column=0, row=7)    #Data Rate
        lbl7.lbl.grid(column=0, row=4)    #Clock Polarity
        lbl8.lbl.grid(column=0, row=8)    #Device Address
        lbl9.lbl.grid(column=0, row=9)    #Chip Select Polarity

        WrRelbl.lbl.grid_forget()    #read write bit for custom packet send (1 bit)
        ComBitlbl.lbl.grid_forget()    #Set command bits (3 bits)
        ResisStatlbl.lbl.grid_forget()    #set resister state (either 0, 1, 2)(4
bits)

        VHin.grid(column=1, row=3)    #Upper Voltage
        cb2.cb.grid(column=1, row=5)    #Frequency
        VLin.grid(column=1, row=2)    #Lower Voltage Level
        cb5.cb.grid(column=1, row=6)    #Device Power
```

```
        cb6.cb.grid(column=1, row=7)    #Data Rates
        cb7.cb.grid(column=1, row=4)    #Clock Polarity
        cb9.cb.grid(column=1, row=9)    #Chip Select Polarity

        AddressText.grid(column=1, row=8)    #Device Address Box

        WrRetxt.grid_forget()    #read write bit for custom packet send (1 bit)
        ComBittxt.grid_forget()    #Set command bits (3 bits)
        ResisStattxt.grid_forget()    #set resister state (either 0, 1, 2)(4 bits)

        btn1.btn.grid(column=2, row=2)    #Send configure

        #Pin Labels
        pin1lbl.lbl.grid(column=0, row=11)    #Create Label for Pin1 dropdown
        pin2lbl.lbl.grid(column=0, row=12)    #Create Label for Pin2 dropdown
        pin3lbl.lbl.grid(column=0, row=13)    #Create Label for Pin3 dropdown
        pin4lbl.lbl.grid(column=0, row=14)    #Create Label for Pin4 dropdown
        pin1spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin2spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin3spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin4spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pinSetBtn.btn.grid(column=2, row=11)    #Send the configuration of the pins
to the board

        #SendReceieve textbox
        B2Rlbl.lbl.grid(column=4, row=2, padx=(50,0))    #Bytes to Receive Label
        B2Rtxt.grid(column=5, row=2)    #Bytes to Receive Input
        Data2Send.lbl.grid(column=4, row=3, padx=(50,0))    #Data to send
        DataText3.grid(column=5, row=3)    #Data to send
        Data2SendBtn.btn.grid(column=6, row=2)    #Send Data from text box

    if cb0.cb.get() == 'Custom Packet':
        lbl1.lbl.grid_forget()    #Upper Voltage
        lbl2.lbl.grid_forget()    #Frequency

        lbl4.lbl.grid_forget()    #Lower Voltage
        lbl5.lbl.grid_forget()    #Device Power
        lbl6.lbl.grid_forget()    #Data Rate
        lbl7.lbl.grid_forget()    #Clock Polarity
        lbl8.lbl.grid_forget()    #Device Address
        lbl9.lbl.grid_forget()    #Chip Select Polarity

        WrRelbl.lbl.grid(column=0, row=1)    #read write bit for custom packet send
(1 bit)
        ComBitlbl.lbl.grid(column=0, row=2)    #Set command bits (3 bits)
        ResisStatlbl.lbl.grid(column=0, row=3)    #set resister state (either 0, 1,
2)(4 bits)

        VHin.grid_forget()    #Upper Voltage
        cb2.cb.grid_forget()    #Frequency
        VLin.grid_forget()    #Lower Voltage Level
        cb5.cb.grid_forget()    #Device Power
        cb6.cb.grid_forget()    #Data Rates
```

```
        cb7.cb.grid_forget()    #Clock Polarity
        cb9.cb.grid_forget()    #Chip Select Polarity


        AddressText.grid_forget()    #Device Address Box

        WrRetxt.grid(column=1, row=1)    #read write bit for custom packet send (1
bit)
        ComBittxt.grid(column=1, row=2)    #Set command bits (3 bits)
        ResisStattxt.grid(column=1, row=3)    #set resister state (either 0, 1,
2)(4 bits)

        btn1.btn.grid(column=2, row=2)    #Send configure


        #Pin Labels
        pin1lbl.lbl.grid_forget()    #Create Label for Pin1 dropdown
        pin2lbl.lbl.grid_forget()    #Create Label for Pin2 dropdown
        pin3lbl.lbl.grid_forget()    #Create Label for Pin3 dropdown
        pin4lbl.lbl.grid_forget()    #Create Label for Pin4 dropdown
        pin1spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin2spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin3spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pin4spi.cb.grid_forget()    #Combobox options for Pin for SPI
        pinSetBtn.btn.grid_forget()    #Send the configuration of the pins to the
board

        #Byte Input Config
        b2lbl.lbl.grid(column = 0, row = 15)    #Input for Byte 2 Label
        b3lbl.lbl.grid(column = 0, row = 16)    #Input for Byte 3 Label
        b4lbl.lbl.grid(column = 0, row = 17)    #Input for Byte 4 Label
        b5lbl.lbl.grid(column = 0, row = 18)    #Input for Byte 5 Label
        byte2in.grid(column = 1, row = 15)    #input for manual packet composition
for Byte 2
        byte3in.grid(column = 1, row = 16)    #input for manual packet composition
for Byte 3
        byte4in.grid(column = 1, row = 17)    #input for manual packet composition
for Byte 4
        byte5in.grid(column = 1, row = 18)    #input for manual packet composition
for Byte 5

        #SendReceieve textbox
        B2Rlbl.lbl.grid_forget()    #Bytes to Receive Label
        B2Rtxt.grid_forget()    #Bytes to Receive Input
        Data2Send.lbl.grid_forget()    #Data to send
        DataText3.grid_forget()    #Data to send
        Data2SendBtn.btn.grid_forget()    #Send Data from text box

cb0.cb.bind("<<ComboboxSelected>>", display_create)    #link protocol selection
combobox to above function. Display fields update when drop box changes.

def main():
    #global ser
```

```
    #ser = serial.Serial('/dev/ttyUSB0')
    InitializePopup()
    #serialThread = threading.Thread(target=read_from_port, args=(ser,))
    #serialThread.start()
    window.mainloop()

if (__name__ == '__main__'):
    main()
```

To make communicating with the I/O Master more streamlined from a computer, development of a library of commands was being created to more easily create and send commands to the I/O Master. The library checks to make sure the data being sent to the I/O master, such as the logic levels being within range, the frequency being within range, and the pin number being a valid pin number. This library is not yet integrated in the GUI, but was planned to be implemented before the final demonstration. (CS)

```
"""
State Commands
"""
def  get_state_command():
    return bytes([0x01, 0x00])

def set_state_configuration_command():
    return bytes([0x01, 0x01])

def set_state_data_command():
    return bytes([0x01, 0x02])

def set_state_run_command():
    return bytes([0x01, 0x03])

def set_state_stop_command():
    return bytes([0x01, 0x04])

"""
Configuration Commands
"""
def get_configuration_target_device_voltage():
    return bytes([0x02, 0x00])

def set_configuration_target_device_voltage(voltage):

def get_configuration_pin_parameters(pinNumber):
    if pinNumber < 1 or pinNumber > 4:
```

```
        raise Exception('pinNumber should be between 1 and 4, found
{}'.format(pinNumber))
    return bytes([0x02, 0x02, pinNumber])

PINPARAMETERS = set(['pullup', 'pulldown', 'tristate', 'disabled'])

def set_configuration_pin_parameters(pinNumber, state):
    if pinNumber < 1 or pinNumber > 4:
        raise Exception('pinNumber should be between 1 and 4, found
{}'.format(pinNumber))
    assert state in PINPARAMETERS
    state_bits = 0b00
    if state == 'pullup':
        state_bits = 0b10
    elif state == 'pulldown':
        state_bits = 0b01
    elif state == 'tristate':
        state_bits = 0b11
    return bytes([0x02, 0x03, pinNumber, state_bits])


def get_configuration_data_speed():
    return bytes([0x02, 0x04])

def set_configuration_data_speed(frequency):
    if frequency < 1 or frequency > 10000:
        raise Exception('frequency should be between 1 and 10000, found
{}'.format(frequency))
    freq_upper = (frequency >> 8) & 0x00FF
    freq_lower = frequency & 0x00FF
    return bytes([0x02, 0x05])


def get_configuration_differential_mode(pinNumber):
    if pinNumber < 1 or pinNumber > 4:
        raise Exception('pinNumber should be between 1 and 4, found
{}'.format(pinNumber))
    return bytes([0x02, 0x06, pinNumber])

def set_configuration_differential_mode(pinNumber, enabled):
    if pinNumber < 1 or pinNumber > 4:
        raise Exception('pinNumber should be between 1 and 4, found
{}'.format(pinNumber))
    if enabled is True:
        return bytes([0x02, 0x07, pinNumber, 0x01])
    elif enabled is False:
        return bytes([0x02, 0x07, pinNumber, 0x00])
    raise Exception('enabled should be either True or False')


def get_configuration_pin_mode(pinNumber):
    if pinNumber < 1 or pinNumber > 4:
```

```
        raise Exception('pinNumber should be between 1 and 4, found
{}'.format(pinNumber))
    return bytes([0x02, 0x08, pinNumber])

PINMODES = set(['disabled', 'input', 'output', 'clock', 'chipselect'])

def set_configuration_pin_mode(pinNumber, mode):
    if pinNumber < 1 or pinNumber > 4:
        raise Exception('pinNumber should be between 1 and 4, found
{}'.format(pinNumber))
    assert mode in PINMODES
    mode_bits = 0b000
    if mode == 'input':
        mode_btis = 0b001
    elif mode == 'output':
        mode_bits = 0b010
    elif mode == 'clock':
        mode_bits = 0b011
    elif mode == 'chipselect':
        mode_bits = 0b100
    return bytes([0x02, 0x09, pinNumber, mode_bits])


def get_configuration_data_parameters(pinNumber):
    if pinNumber < 1 or pinNumber > 4:
        raise Exception('pinNumber should be between 1 and 4, found
{}'.format(pinNumber))
    return bytes([0x02, 0x0A, pinNumber])

def set_configuration_data_parameters(pinNumber, idleState, polarity):
    if pinNumber < 1 or pinNumber > 4:
        raise Exception('pinNumber should be between 1 and 4, found
{}'.format(pinNumber))
    idleState_bits = 0b00
    if idleState is True:
        idleState_bits = 0b10
    elif idleState is not False:
        raise Exception('idleSate should be True or False')
    polarity_bits = 0b00
    if polarity is True:
        polarity_bits = 0b01
    elif polarity is not False:
        raise Exception('polarity should be True or False')
    return bytes([0x02, 0x0B, pinNumber, idleState_bits | polarity_bits])


def get_configuration_pin_logic_levels(pinNumber):
    return bytes([0x02, 0x0C])

def set_configuration_pin_logic_levels(pinNumber, highVoltage, lowVoltage):
    if pinNumber < 1 or pinNumber > 4:
        raise Exception('pinNumber should be between 1 and 4, found
{}'.format(pinNumber))
```

```
    if highVoltage < lowVoltage or highVoltage > 15 or highVoltage < 0 or
lowVoltage > 15 or lowVoltage < 0:
        raise Exception('highVoltage must be >= lowVoltage and both must be in
range of [0, 15]')

    high_voltage_scaled = round(((2**12 - 1) / 33) * (16.5 - VH))
    low_voltage_scaled = round(((2**12 - 1) / 33) * (16.5 - VL))

    high_voltage_upper_bits = (high_voltage_scaled >> 8) & 0x00FF
    high_voltage_lower_bits = high_voltage_scaled & 0x00FF

    low_voltage_upper_bits = (low_voltage_scaled >> 8) & 0x00FF
    low_voltage_lower_bits = low_voltage_scaled & 0x00FF

    return bytes([0x02, 0x0D, pinNumber, high_voltage_upper_bits,
high_voltage_lower_bits, low_voltage_upper_bits, low_voltage_lower_bits])


"""
Data Commands
"""
def set_data_send(data):
    size = len(data)
    if size > 255 or size <= 0:
        raise Exception('Can only support [1,255] bytes of data, found {}
bytes'.format(size))
    return bytearray([0x03, 0x00, size]).append(data)

def set_data_receive(numBytes):
    if numBytes <= 0 or numBytes > 255:
            raise Exception('Can only support [1,255] bytes of data, found {}
bytes'.format(numBytes))
    return bytes([0x03, 0x01, numBytes])
```

6. Financial Budget

The total expenses for this project are listed in Table 57.

Table 57: Project Expenditures

Beginning Balance			**$600.00**
2019-10-03	DigiKey	STM32H7 Development Board	-$27.00
2020-02-07	DigiKey	I/O Master v1.0 BOM	-$280.86
2020-02-13	OSH Park	I/O Master v1.0 PCBs	-$263.70
Ending Balance			**$28.44**

7. Project Schedule

The proposed project schedule for implementation is listed in Table 58. The actual project implementation dates are listed in Table 59.

Table 58: Proposed Project Implementation Schedule

Task	Duration	Start Date	Finish Date	Assignees
v1.0 Hardware Implementation	36 days	2020-01-13	2020-03-01	
Schematic Design	6 days	2020-01-23	2020-01-30	Ian Glen Nik Untch
Design Review (In-Lab)	1 day	2020-01-30	2020-01-30	Aaron Dubler Corey Dye Cameron Sinko Ian Glen Nik Untch
PCB Design	3 days	2020-01-30	2020-02-03	Ian Glen Nik Untch
Create BOM	1 day	2020-02-01	2020-02-01	Ian Glen Nik Untch
Order PCB	6 days	2020-02-03	2020-02-10	Ian Glen
Build PCB	5 days	2020-02-10	2020-02-14	Ian Glen Nik Untch
Test/Troubleshoot PCB	6 days	2020-02-14	2020-02-21	Aaron Dubler Corey Dye Cameron Sinko Ian Glen Nik Untch
v1.0 Software Implementation	36 days	2020-01-13	2020-03-01	
Configuration Command Scheme	9 days	2020-01-13	2020-01-23	Aaron Dubler
MCU Configuration Function Parsing	14 days	2020-01-13	2020-01-30	Aaron Dubler
SPI GUI Window Complete	14 days	2020-01-13	2020-01-30	Corey Dye
USB NCM Configuration	14 days	2020-01-13	2020-01-30	Cameron Sinko
USB & I/O Testing and Analysis	6 days	2020-01-30	2020-02-06	Cameron Sinko
Configuration MCU Pin Implementation	6 days	2020-01-30	2020-02-06	Aaron Dubler
GUI Class for Protocol Configuration	11 days	2020-01-30	2020-02-13	Corey Dye
Python Configuration Functions	6 days	2020-02-06	2020-02-13	Corey Dye

I/O Data Revisions	11 days	2020-02-06	2020-02-20	Cameron Sinko
Testing Python and MCU Configuration	6 days	2020-02-13	2020-02-20	Aaron Dubler Corey Dye
Initial Code Review	6 days	2020-02-20	2020-02-27	Aaron Dubler Corey Dye Cameron Sinko
v2.0 Hardware Implementation	60 days	2020-02-20	2020-04-20	
Schematic Design	6 days	2020-02-20	2020-02-27	Ian Glen Nik Untch
Design Review (In-Lab)	1 day	2020-02-27	2020-02-27	Aaron Dubler Corey Dye Cameron Sinko Ian Glen Nik Untch
PCB Design	3 days	2020-02-27	2020-03-01	Ian Glen Nik Untch
Create BOM	1 day	2020-03-01	2020-03-01	Ian Glen Nik Untch
Order PCB	9 days	2020-03-01	2020-03-09	Ian Glen
Build PCB	6 days	2020-03-09	2020-03-16	Ian Glen Nik Untch
Test/Troubleshoot PCB	32 days	2020-03-16	2020-04-17	Aaron Dubler Corey Dye Cameron Sinko Ian Glen Nik Untch
v2.0 Software Implementation	53 days	2020-02-27	2020-04-20	
Fix v1.0 MCU and GUI Bugs	6 days	2020-02-27	2020-03-05	Aaron Dubler Corey Dye Cameron Sinko
RS-485 GUI Window	6 days	2020-03-05	2020-03-12	Corey Dye
RS-485 I/O	6 days	2020-03-05	2020-03-13	Aaron Dubler Cameron Sinko
Test RS-485 System	6 days	2020-03-12	2020-03-19	Aaron Dubler Corey Dye Cameron Sinko
UART GUI Window	6 days	2020-03-19	2020-03-26	Corey Dye
UART I/O	6 days	2020-03-19	2020-03-26	Aaron Dubler Cameron Sinko

I2C GUI Window	6 days	2020-03-26	2020-04-02	Corey Dye
I2C I/O	6 days	2020-03-26	2020-04-02	Aaron Dubler Cameron Sinko
SWD GUI Window	6 days	2020-04-02	2020-04-09	Corey Dye
SWD I/O	6 days	2020-04-02	2020-04-09	Aaron Dubler Cameron Sinko
Testing / Fix Bugs	8 days	2020-04-09	2020-04-17	Aaron Dubler Corey Dye Cameron Sinko Ian Glen Nik Untch

Table 59: Actual Project Implementation Dates

Team	**Task**	**Start Date**	**Finish Date**	**Assignees**
v1.0 Hardware	Schematic Design	2020-01-13	2020-02-04	Ian Glen Nik Untch
Software	Configuration Command Scheme	2020-01-13	2020-01-30	Aaron Dubler
Software	MCU Configuration Function Parsing	2020-01-13	2020-01-30	Aaron Dubler
Software	SPI GUI Window Complete	2020-01-13	2020-01-30	Corey Dye
v1.0 Hardware	Design Review (In-Lab)	2020-02-13	2020-02-13	Aaron Dubler Corey Dye Cameron Sinko Ian Glen Nik Untch
v1.0 Hardware	PCB Design	2020-02-04	2020-02-13	Ian Glen Nik Untch
v1.0 Hardware	Create BOM	2020-02-12	2020-02-12	Ian Glen Nik Untch
Software	Configuration MCU Pin Implementation	2020-01-30	2020-02-20	Aaron Dubler
Software	USB NCM Configuration	2020-01-13	2020-02-20	Cameron Sinko
Software	USB & I/O Testing and Analysis	2020-01-30	2020-02-20	Cameron Sinko
Software	GUI Class for Protocol Configuration	2020-01-30	2020-02-20	Corey Dye
Software	Python Configuration Functions	2020-02-06	2020-02-20	Corey Dye
v1.0 Hardware	v1.0 Hardware - Order PCB	2020-02-13	2020-02-27	Ian Glen
v1.0 Hardware	v1.0 Hardware - Build PCB	2020-02-27	2020-03-03	Ian Glen Nik Untch
Software	Fix v1.0 MCU and GUI Bugs		2020-03-09	Aaron Dubler Corey Dye Cameron Sinko

Both	Hardware & Software Testing	2020-03-03	2020-03-09	Aaron Dubler Corey Dye Cameron Sinko Ian Glen Nik Untch
Mid-term Demo		2020-03-05	2020-03-05	
In-Person Classes Cancelled - Work Halted		2020-03-09		

8. Team Information

- Aaron Dubler, *CpE*
- Corey Dye, *CpE*
- Ian Glen, *EE, Not taken ESI* **Team Lead, Hardware Lead**
- Cameron Sinko, *CpE* **Software Lead**
- Nikolaus Untch, *EE, Currently taking ESI* **Data Manager**

9. Conclusions and Recommendations

Designing a general purpose device with capabilities of interfacing at high frequencies of up to 10 MHz, support for differential logic, and making it cost effective is a challenge. Through selecting components with specifications significantly higher than the requirements, A high speed system can be designed with room for further optimizations in both cost cutting and higher performance for the future. While purely sending and receiving data is very achievable with our design and research, there are a few aspects that the design method overlooked.

One very important aspect to consider for the design of a high speed device is propagation delay, which was not considered. In order to send and receive data at 10 MHz, components with support for a 20 MHz signal is necessary. This means the rise time, fall time, and propagation delay must be around 50 nanoseconds in order to get the signal through the devices. Without careful design and consideration, it is easy to go over 50 nanoseconds. The delays of multiplexers within the microcontroller even have an effect on speed, which limits I/O to a maximum of 30 MHz. In many cases, this data is not available on datasheets, so real world testing must be done to determine the delay for some components of the circuit.

Another aspect to more closely consider is how to quantify safety for overcurrent and overvoltage protection. In order for a circuit to offer any true benefit, the amount of time it must take for these protections to kick in must be considered. While there are designs for safety circuits to handle these dangers, it is unknown as to how well they can protect other devices from actually breaking. Suppose a user accidentally shorted a high voltage signal that was not meant to run through the target device, there needs to be a quantification as to how much time must

elapse before there is actual damage to most devices. Again, real world testing may be the easiest way to determine the answer to such a question.

In terms of working on the I/O Master in a team of five, there was certainly enough work for five people. The project was organized in such a way that each person generally only needed to talk to 1 or 2 other people on the team in order to design and implement their parts. On one side, Nik designed solely electrical characteristics for the board. Ian designed the embedded portions of the electronics, including the microcontroller layout, which required dabbling into software features. Cameron had to write the lower level firmware and create a project to bring up the custom board and get it to work, which required some hardware knowledge. Aaron had to write software to interface with the selected pins in the design, along with communicating with a computer, and Corey had to interface with the control code Aaron was writing which was purely software. Each individual did not know much about the entire project beyond the level 1 block diagram, but together as a team, we were able to design a large, complex system that has many detailed parts. As a result, we had to put trust into other team members that they would be able to design and implement something that would satisfy the high level requirements for integration.

The project was overall very successful and everyone learned a lot of skills about design and implementation. By choosing a project that we all knew we needed to do legitimate research on to figure out how to build it, we were able to come up with several metrics and really think about if the design would meet the metrics such as current limits, voltage rails for scaling, and frequency division. We were definitely on track to fully implementing all of the features we planned without needing to do any significant redesign of any components. By spending the extra

time early on developing a solid base for the implementation of features, the work produced is high quality, easy to modify and expand upon, and breaks the large parts of the design into smaller, easier to understand pieces.

Looking forward, some team members are planning to finish getting the prototype functional over the summer and go back to evaluate adding additional features to the I/O Master such as the ability to save data to an SD card, support for wireless communication with a computer, and a complete library to write programs for interfacing with it. There is also consideration in writing the firmware in Rust, an emerging programming language that focuses on memory safety and speed optimizations by design.

For future students, we highly recommend trying to get familiar with the process of designing a board, assembling it, and getting software onto it for the first time. Because we were familiar with all of this from past experience, we were able to plan to get the board assembled and functional fairly quickly, and it was still several days of testing and fixing solder joints before it was functional. The faster a PCB can be produced for the project, the more iterations the project can go through before final demonstrations, which is very valuable.

10. References

[1] Leens, F., “An introduction to I2C and SPI protocols”, *Instrumentation & Measurement Magazine*, IEEE , vol.12, no.l, pp.8-13, February 2009. [Online]. Available: https://ieeexplore.ieee.org/document/4762946

[2] National Instruments Corporation, “Digital States, Voltage Levels, and Logic Families” [Online]. Available: http://www.ni.com/white-paper/3292/en/.

[3] Microchip Technology, Inc., “PIC24FXXKA1XX/FVXXKA3XX Family Flash Programming Specifications” [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/DS-39919c.pdf

[4] “Bus Pirate”. Dangerous Prototypes. http://dangerousprototypes.com/docs/Bus_Pirate

[5] M. Ossmann., “GreatFET: Making GoodFET Great Again” [Online]. Available: https://www.blackhat.com/docs/us-16/materials/us-16-Ossmann-GreatFET-Making-GoodFET-Great-Again-wp.pdf

[6] O. Rancu and M. Moldovan. (2005). *U.S. Patent No.* 7,099,438. Washington, DC: U.S. Patent and Trademark Office. [Online]. Available: https://patentimages.storage.googleapis.com/17/88/37/8050f53426ff47/US7099438.pdf

[7] N. Stong, “Universal serial bus (USB) to universal interface using field programmable gate arrays (FPGA) to mimic traditional hardware [military aircraft testing applications]”, AUTOTESTCON 2003, IEEE Systems Readiness Technology Conference., Proceedings, 22-25 Sept. 2003, pp. 386 - 391. [Online]. Available: https://ieeexplore.ieee.org/document/1243602

[8] R. Castro. (2004). *U.S. Patent No.* 7,649,726. Washington, DC: U.S. Patent and Trademark Office. [Online]. Available: https://patentimages.storage.googleapis.com/04/58/a8/32f41986567530/US7649726.pdf

[9] Kok Seng Ting, Gee Keng Ee, Chee Kyun Ng, N.K. Noordin and B.M. Ali,"The performance evaluation of IEEE 802.11 against IEEE 802.15.4 with low transmission power," 17th Asia-Pacific Conference on Communications (APCC), pp.850,855, 2-5 Oct. 2011. Available: https://ieeexplore.ieee.org/document/6152927

[10] NXP Semiconductors N.V., “UM10204 I2C-bus specification and user manual” [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[11] VTI Technologies, Inc., “TN15 SPI Interface Specification” [Online]. Available: https://www.mouser.com/pdfdocs/tn15_spi_interface_specification.PDF

[12] Texas Instruments, Inc., “Interface Circuits for TIA/EIA-232-F” [Online]. Available: https://www.ti.com/lit/an/slla037a/slla037a.pdf

[13] Texas Instruments, Inc., “RS-422 and RS-485 Standards Overview and System Configurations” [Online]. Available: https://www.ti.com/lit/an/slla070d/slla070d.pdf

[14] 32-bit MCU table http://ww1.microchip.com/downloads/en/DeviceDoc/60001455D.pdf

[15] pic32mzda family datasheet http://ww1.microchip.com/downloads/en/DeviceDoc/PIC32MZ_DA%20_Family%20Datasheet_DS60001361H.pdf

11. Appendices

11.1 I/O Master v1.0 Schematic

TP4
KEYSTONE 5011
GND

TP16
Vin
F1
0.5A
D1
B130L
D2
B130L
TP17
Q27
DMP3099L
R262
100K
J1
54-00166
OPTIONAL DC INPUT
(12V, 500mA)
GND
TP18
TP19
J2
10118193
USB 2.0 INPUT
(5V, 500mA)
VBUS
D+
D-
ID
Shield
GND
C2
1uF
C1 0.47u
CHASSIS
GND

INPUT: 4.6-15V
U1
AZ1117-3.3
Vin
VI
VO
GND
C3
0.1u
C4
10u
C5
22u
C6
0.1u
3.3V
3.3V REGULATOR

Sheet: 15V Regulator
Vin
+/-15V
REGULATOR
+15V
-15V
File: 15V Regulator.sch

POWER SUPPLY LEDs
3.3V
+15V
-15V
Vout
D3
ORANGE
D4
BLUE
D5
BLUE
D6
GREEN
R5
1K
R6
10K
R7
10K
R8
4.7K
GND

Sheet: Microcontroller
3.3V
USB_VBUS
USB_D+
USB_D-
I2C_SDA
I2C_SCL
IO_1_OUT
IO_1_IN
IO_1_TRIS
IO_1_PU
IO_1_PD
IO_2_OUT
IO_2_IN
IO_2_TRIS
IO_2_PU
IO_2_PD
IO_1&2_DIFF
IO_1&2_TERM
IO_3_OUT
IO_3_IN
IO_3_TRIS
IO_3_PU
IO_3_PD
IO_4_OUT
IO_4_IN
IO_4_TRIS
IO_4_PU
IO_4_PD
IO_3&4_DIFF
IO_3&4_TERM
Vout_SETPOINT
MICROCONTROLLER
File: Microcontroller.sch

Sheet: Level Shifter - I/O Pins 1&2
3.3V
+15V
-15V
I2C_SDA
I2C_SCL
IO_1_OUT
IO_1_IN
IO_1_TRIS
IO_1_PU
IO_1_PD
IO_2_OUT
IO_2_IN
IO_2_TRIS
IO_2_PU
IO_2_PD
IO_1&2_DIFF
IO_1&2_TERM
LEVEL SHIFTER
File: Level Shifter - IO 1-2.sch

Sheet: Level Shifter - I/O Pins 3&4
3.3V
+15V
-15V
I2C_SDA
I2C_SCL
IO_3_OUT
IO_3_IN
IO_3_TRIS
IO_3_PU
IO_3_PD
IO_4_OUT
IO_4_IN
IO_4_TRIS
IO_4_PU
IO_4_PD
IO_3&4_DIFF
IO_3&4_TERM
LEVEL SHIFTER
File: Level Shifter - IO 3-4.sch

Sheet: Circuit Protection - I/O Pin 1
+15V
-15V
IO_1
CIRCUIT
PROTECTION
TARGET_IO_1
File: Circuit Protection - IO 1.sch

Sheet: Circuit Protection - I/O Pin 2
+15V
-15V
IO_2
CIRCUIT
PROTECTION
TARGET_IO_2
File: Circuit Protection - IO 2.sch

Sheet: Circuit Protection - I/O Pin 3
+15V
-15V
IO_3
CIRCUIT
PROTECTION
TARGET_IO_3
File: Circuit Protection - IO 3.sch

Sheet: Circuit Protection - I/O Pin 4
+15V
-15V
IO_4
CIRCUIT
PROTECTION
TARGET_IO_4
File: Circuit Protection - IO 4.sch

Sheet: Adjustable Regulator
3.3V
Vin
SETPOINT
3.3 - 15V
ADJUSTABLE
REGULATOR
Vout
File: Adjustable Regulator.sch

Vout
F2
0.1A
J3
I/O PIN 1
I/O PIN 2
I/O PIN 3
I/O PIN 4
Vout
0.1" HDR
GND
I/O PINS
TO TARGET DEVICE

PWR_FLAG
Vin
+15V
-15V
CHASSIS

DO NOT POP
H1
H2
H3
H4
CHASSIS

DT18 - I/O MASTER
THE UNIVERSITY OF AKRON
Sheet: /
File: io-master.sch
Title: I/O MASTER v1.0
Size: B
Date: 2020-02-13
Rev: A
KiCad E.D.A. kicad 5.1.5
Id: 1/10

STM32H7 MICROCONTROLLER

NOTE: DECOUPLING CAPS, PLACE CLOSE TO VDD

3.3V — C8 4.7u, C16 0.1u, C19 0.1u, C22 0.1u, C24 0.1u, C26 0.1u — GND

3.3V — C9 0.1u, C17 0.1u, C20 0.1u, C23 0.1u, C25 0.1u, C27 0.1u — GND

NOTE: DECOUPLING CAPS, PLACE CLOSE TO VDD33_USB

3.3V — C10 1u — GND

NOTE: DECOUPLING CAPS, PLACE CLOSE TO VDDA

3.3V — C11 1u, C15 0.1u — GND

NOTE: PLACE CRYSTAL CLOSE TO OSC_OUT/OSC_IN

C12 20p, Y1 48MHz, C13 20p, R9 DO NOT POP — OSC_OUT, OSC_IN — GND

NOTE: DECOUPLING CAPS, PLACE CLOSE TO VCAP1/VCAP2

VCAP — C7 2.2u, C14 2.2u — GND

JTAG/SERIAL WIRE DEBUG

J4 0.1" HDR — 3.3V, GND, GND, KEY, GNDdetect

VCC	1	2	TMS_SWDIO — SWD I/O
GND	3	4	TCK_SWDCLK — SWD CLOCK
GND	5	6	TDO_SWO — SWD TRACE
KEY	7	8	TDI — JTAG DATA IN
GNDdetect	9	10	nRESET — JTAG RESET

IO_1_OUT, IO_2_OUT, IO_1_TRIS, IO_2_TRIS, IO_1&2_DIFF, IO_1_PU, IO_2_PU, IO_3_OUT, IO_4_OUT, IO_3_TRIS, IO_4_TRIS, IO_3&4_DIFF, IO_3_PU, IO_4_PU

R10–R42 100K

I2C_SCL, I2C_SDA, IO_1_PD, IO_2_PD, IO_1&2_TERM, IO_3_PD, IO_4_PD, IO_3&4_TERM

R16 4.7K, R18 4.7K, R21–R37 100K

IO_1_TRIS, IO_1_IN, IO_2_TRIS, IO_2_IN, IO_3_TRIS, IO_3_IN, IO_4_TRIS, IO_4_IN

R11–R13, R19–R22, R30–R33, R39–R40 4.7K

D7, D8, D9, D11 YELLOW/GREEN

IO_1_OUT, IO_2_OUT, IO_3_OUT, IO_4_OUT — GND

RGB STATUS LED

STATUS_R — R27 330
STATUS_G — R28 330
STATUS_B — R29 330
D10 RGB — GND

RESET BUTTON

SW2 EVQ-Q2B03W, C28 0.1u — GND

BOOT SWITCH

3.3V — SW1 JS202011SCQN — R43 10K — GND

FB1 MZ1608-102Y — 3.3V — VDDA

U2 STM32H743ZITx

		PA0	34 — IO_1_OUT
		PA1	35 — IO_2_OUT
		PA2	36 — IO_3_OUT
		PA3	37 — ULPI_D0
		PA4	40 — Vout_SETPOINT
		PA5	41 — ULPI_CK
		PA6	42 — IO_6_PU — TP22
		PA7	43 — IO_8_OUT
		PA8	100 — I2C_SCL
		PA9	101 — UART_TXO — TP82
		PA10	102 — UART_RXI — TP83
		PA11	103 — USB_FS_DM — TP84
		PA12	104 — USB_FS_DP — TP85
		PA13	105 — TMS_SWDIO
		PA14	109 — TCK_SWDCLK
		PA15	110 — TDI
		PB0	46 — ULPI_D1
		PB1	47 — ULPI_D2
		PB2	48 — IO_6_PD — TP23
		PB3	133 — TDO_SWO
		PB4	134 — nRESET
		PB5	135 — ULPI_D7
		PB6	136 — IO_4_PU
		PB7	137 — IO_5_OUT
		PB8	139 — IO_6_OUT
		PB9	140 — IO_7_OUT
		PB10	69 — ULPI_D3
		PB11	70 — ULPI_D4
		PB12	73 — ULPI_D5
		PB13	74 — ULPI_D6
		PB14	75 — IO_1_PU
		PB15	76 — IO_1_PD
		PC0	26 — ULPI_STP
		PC1	27 — IO_7_PU — TP24
		PC2_C	28 — ULPI_DIR
		PC3_C	29 — ULPI_NXT
		PC4	44 — IO_7_PD — TP25
		PC5	45 — IO_8_PU — TP26
		PC6	96 — IO_4_OUT
		PC7	97 — IO_8_PD — TP27
		PC8	98 — IO_7&8_TERM — TP90
		PC9	99 — I2C_SDA
		PC10	111 — IO_3&4_TERM
		PC11	112 — IO_3&4_DIFF
		PC12	113 — IO_4_PD
		PC13	7 — IO_7&8_DIFF — TP89
		PC14	8 — IO_5&6_DIFF — TP91
		PC15	9 — IO_5&6_TERM — TP92
		PD0	114 — IO_1_OUT
		PD1	115 — IO_2_OUT
		PD2	116 — IO_3_OUT
		PD3	117 — IO_4_OUT
		PD4	118 — IO_5_OUT — TP93
		PD5	119 — IO_6_OUT — TP94
		PD6	122 — IO_7_OUT — TP95
		PD7	123 — IO_8_OUT — TP96
		PD8	77 — IO_1_TRIS
		PD9	78 — IO_2_TRIS
		PD10	79 — IO_3_TRIS
		PD11	80 — IO_4_TRIS
		PD12	81 — IO_5_TRIS — TP97
		PD13	82 — IO_6_TRIS — TP98
		PD14	85 — IO_7_TRIS — TP99
		PD15	86 — IO_8_TRIS — TP100
VBAT	6	PE0	141
PDR_ON	143	PE1	142
BOOT0	138	PE2	1
VCAP1	71	PE3	2
VCAP2	106	PE4	3
VREF+	32	PE5	4
OSC_OUT	23	PE6	5
OSC_IN	24	PE7	58
PH0		PE8	59
PH1		PE9	60
IO_5_PU	56	PE10	63
IO_5_PD	57	PE11	64
IO_1&2_DIFF	87	PE12	65 — IO_5_IN
IO_1&2_TERM	88	PE13	66 — IO_6_IN
IO_2_PD	89	PE14	67 — IO_7_IN
IO_2_PU	90	PE15	68 — IO_8_IN
IO_3_PU	91		
IO_3_PD	92		
STATUS_B	10		
STATUS_R	11		
STATUS_G	12		
VDD	17, 30, 39, 52, 62, 72, 84, 108, 121, 131, 144, 95		
VDDA	33		
VDD33_USB	33		
VSS	16, 38, 51, 61, 83, 94, 107, 120, 130, 31, VSSA		

TP20 IO_5_PU, TP21 IO_5_PD, TP28–TP35, TP36–TP48, TP49–TP60

TP61 IO_1_OUT, TP62 IO_2_OUT, TP63 IO_3_OUT, TP64 IO_4_OUT

TP69 IO_1_TRIS, TP70 IO_2_TRIS, TP71 IO_3_TRIS, TP72 IO_4_TRIS

TP77 IO_1_PU, TP78 IO_2_PU, TP79 IO_3_PU, TP80 IO_4_PU

TP104 IO_1_PD, TP105 IO_2_PD, TP106 IO_3_PD, TP107 IO_4_PD

TP65 IO_1_IN, TP66 IO_2_IN, TP67 IO_3_IN, TP68 IO_4_IN

TP73 IO_1&2_DIFF, TP74 IO_1&2_TERM, TP75 IO_3&4_DIFF, TP76 IO_3&4_TERM

TP101 I2C_SCL, TP102 I2C_SDA, TP103 Vout_SETPOINT

MCU ALTERNATE PIN FUNCTIONS:
- PA0: TIM2_CH1
- PA1: TIM5_CH2
- PA2: TIM15_CH1
- PA4: DAC_OUT1
- PA7: TIM14_CH1
- PA8: I2C1_SCL
- PA9: USART1_TX
- PA10: USART1_RX
- PB7: TIM4_CH2
- PB8: TIM16_CH1
- PB9: TIM17_CH1
- PC6: TIM3_CH1
- PC9: I2C1_SDA

USB 2.0 HIGH SPEED PHY

NOTE: DECOUPLING CAPS, PLACE CLOSE TO VDD3.3

3.3V — C31 0.1u, C35 0.1u, C36 0.1u, C38 0.1u — GND

U3 USB3300-EZK

3.3V — VDD33 (6, 16, 25, 30)

USB_VBUS — R44 820 — D12 DDZ5V6ASF — GND

VBUS	4	USB_VBUS
DM	8	USB_D−
DP	7	USB_D+
ID	5	
XO	27	
XI	28	
CPEN	3	
EXTVBUS	10	
REG_EN	31	3.3V
RBIAS	32	R45 12K — GND
GND	1, 2, 33	GND

VDD1.8	15, 26	VDD1.8
VDDA1.8	29	VDDA1.8
DATA7	17	ULPI_D7
DATA6	18	ULPI_D6
DATA5	19	ULPI_D5
DATA4	20	ULPI_D4
DATA3	21	ULPI_D3
DATA2	22	ULPI_D2
DATA1	23	ULPI_D1
DATA0	24	ULPI_D0
NXT	11	ULPI_NXT
DIR	12	ULPI_DIR
STP	13	ULPI_STP
CLKOUT	14	ULPI_CK
RESET	9	TP174

NOTE: PLACE CRYSTAL CLOSE TO XO/XI

C33 20p, Y2 24MHz, C34 20p, R46 1M — XO, XI — GND

NOTE: DECOUPLING CAPS, PLACE CLOSE TO VDD1.8

VDD1.8 — C29 4.7u, C30 0.1u, C32 0.1u — GND

NOTE: DECOUPLING CAPS, PLACE CLOSE TO VDDA1.8

VDDA1.8 — C37 4.7u, C39 0.1u — GND

DT18 – I/O MASTER
THE UNIVERSITY OF AKRON

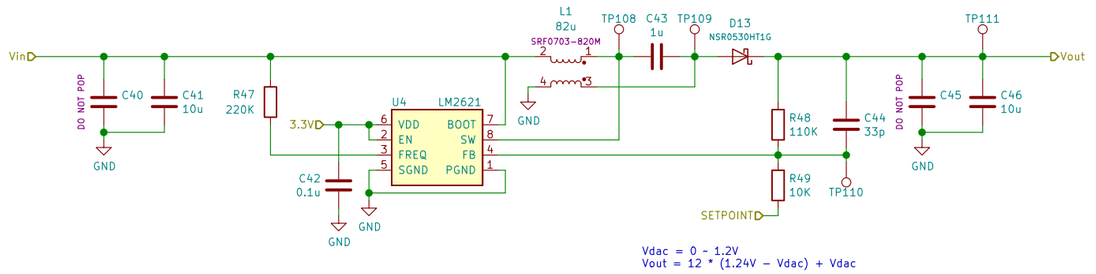
Sheet: /Microcontroller/
File: Microcontroller.sch

Title: MICROCONTROLLER SUBSYSTEM

Size: B	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 2/10

3.3–15V ADJUSTABLE REGULATOR

INPUT
(1.2–14V)



TARGET DEVICE SUPPLY
(3.3–15V ADJ, 100mA MAX)

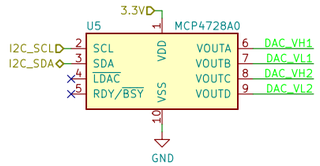
Vdac = 0 ~ 1.2V
Vout = 12 * (1.24V – Vdac) + Vdac

DT18 – I/O MASTER
THE UNIVERSITY OF AKRON

Sheet: /Adjustable Regulator/	
File: Adjustable Regulator.sch	
Title: 3.3–15V ADJUSTABLE REGULATOR	
Size: B	Date: 2020–02–13
KiCad E.D.A. kicad 5.1.5	Id: 3/10

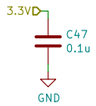
Rev: A

LOGIC LEVEL GENERATOR

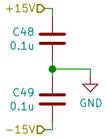


NOTE: P/N MCP4728A0 IS PRE-PROGRAMMED WITH I2C ADDRESS 0b1100000

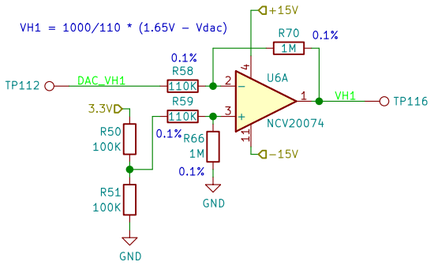
NOTE: DECOUPLING CAP, PLACE CLOSE TO VDD

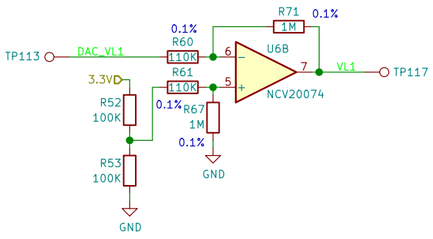


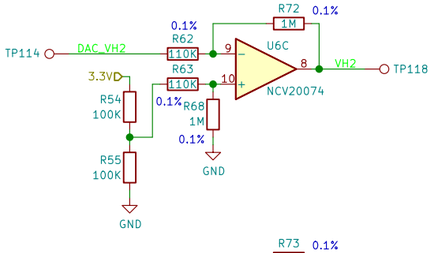
NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+/V-

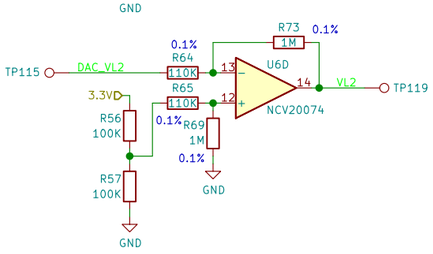


VH1 = 1000/110 * (1.65V - Vdac)



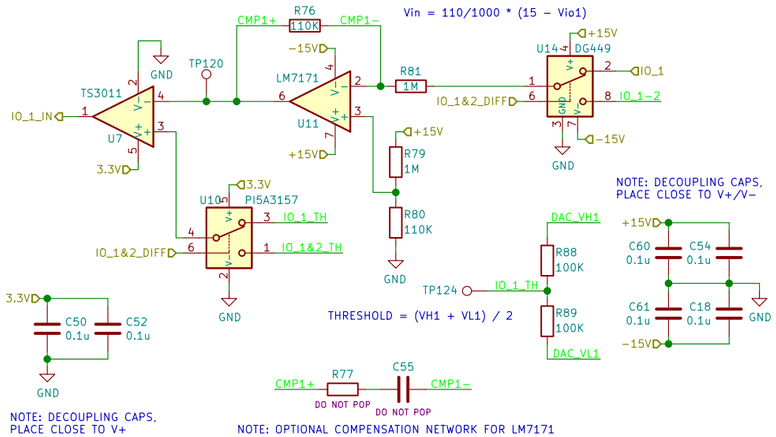






SINGLE-ENDED RECEIVER (I/O PIN 1)

Vin = 110/1000 * (15 - Vio1)



NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+/V-

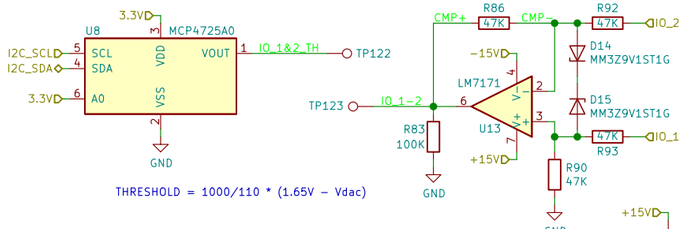
THRESHOLD = (VH1 + VL1) / 2

NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+

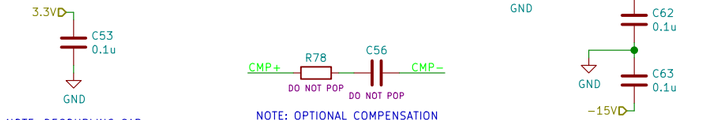
NOTE: OPTIONAL COMPENSATION NETWORK FOR LM7171

DIFFERENTIAL RECEIVER (I/O PIN 1 - I/O PIN 2)

NOTE: P/N MCP4725A0 IS PRE-PROGRAMMED WITH I2C ADDRESS 0b1100001



THRESHOLD = 1000/110 * (1.65V - Vdac)



NOTE: DECOUPLING CAP, PLACE CLOSE TO VDD

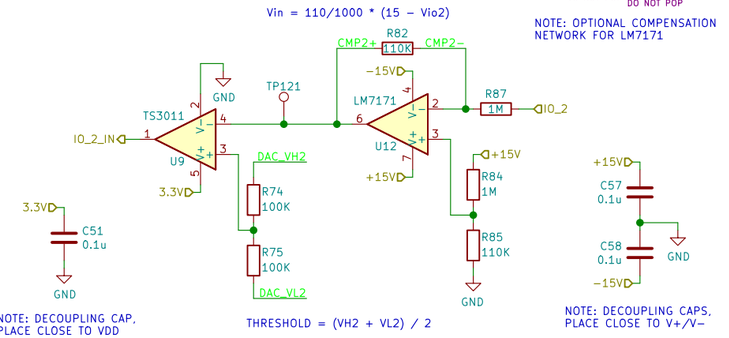
NOTE: OPTIONAL COMPENSATION NETWORK FOR LM7171

NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+/V-

SINGLE-ENDED RECEIVER (I/O PIN 2)

NOTE: OPTIONAL COMPENSATION NETWORK FOR LM7171

Vin = 110/1000 * (15 - Vio2)

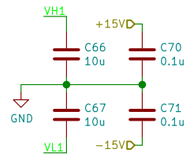


NOTE: DECOUPLING CAP, PLACE CLOSE TO VDD

THRESHOLD = (VH2 + VL2) / 2

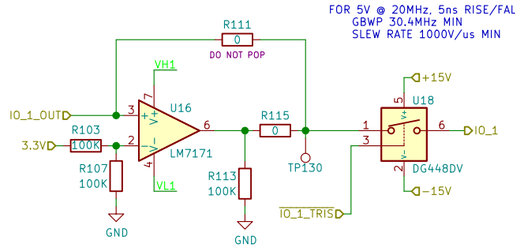
NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+/V-

OUTPUT DRIVER (I/O PIN 1)

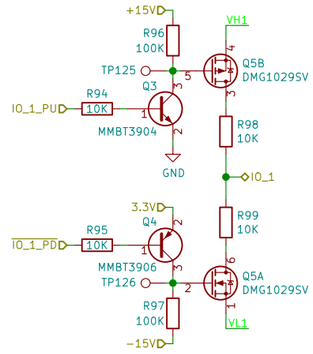


NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+/V-

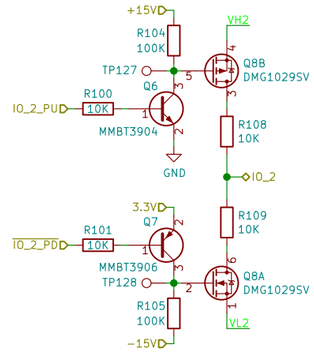
FOR 5V @ 20MHz, 5ns RISE/FALL:
GBWP 30.4MHz MIN
SLEW RATE 1000V/us MIN



CONFIGURABLE RESISTORS

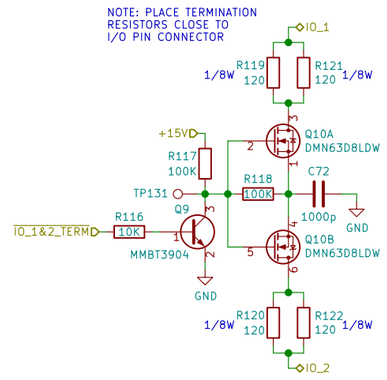


I/O PIN 1
10KΩ PULL-UP/DOWN



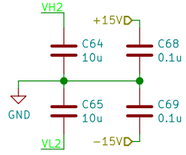
I/O PIN 2
10KΩ PULL-UP/DOWN

NOTE: PLACE TERMINATION RESISTORS CLOSE TO I/O PIN CONNECTOR



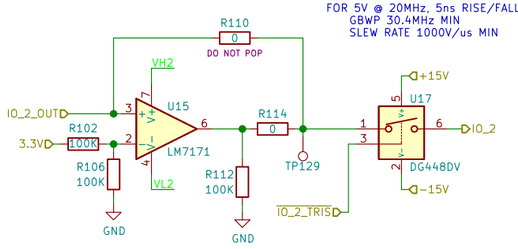
120Ω TERMINATION
BETWEEN I/O PINS 1 & 2

OUTPUT DRIVER (I/O PIN 2)



NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+/V-

FOR 5V @ 20MHz, 5ns RISE/FALL:
GBWP 30.4MHz MIN
SLEW RATE 1000V/us MIN



DT18 - I/O MASTER
THE UNIVERSITY OF AKRON

Sheet: /Level Shifter - I/O Pins 1&2/
File: Level Shifter - IO 1-2.sch

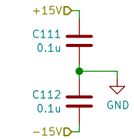
Title: LEVEL SHIFTER SUBSYSTEM - I/O PINS 1 & 2

Size: B	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 4/10

OVERVOLTAGE PROTECTION

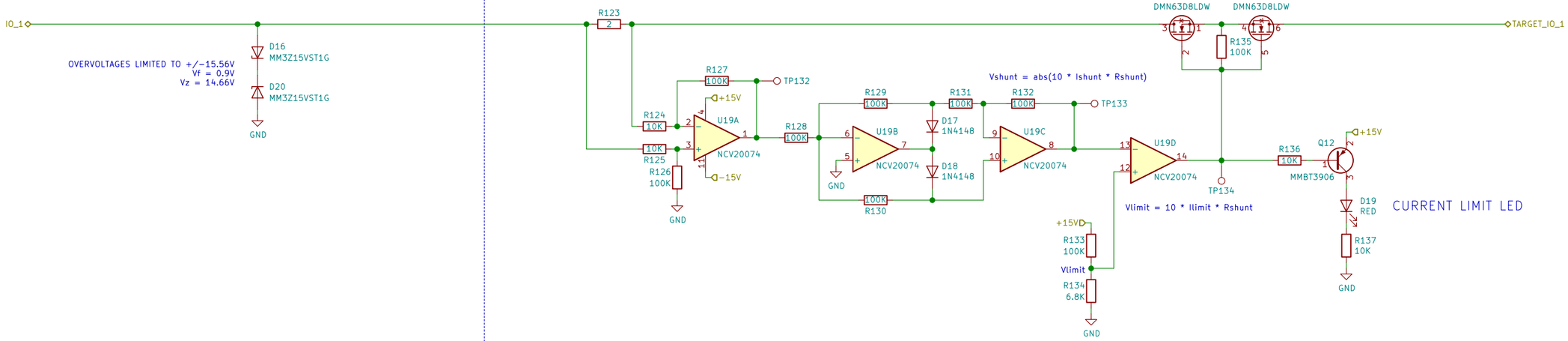
OVERCURRENT PROTECTION

NOTE: DECOUPLING CAPS,
PLACE CLOSE TO V+/V-



TRANSISTOR POWER DISSIPATION:
(WORST CASE: OUPUTTING +15V, PIN SHORTED TO -15V)
Pq = (15V - -15V) * 50mA = 1.5W

MAX AVG Pq IS 275mW.
ALLOWABLE DUTY CYCLE IS 0.275 / 1.5 = 0.18
(10 SEC SHORT CIRCUIT OVER 1 MIN)



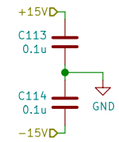
DT18 - I/O MASTER
THE UNIVERSITY OF AKRON

Sheet: /Circuit Protection - I/O Pin 1/		
File: Circuit Protection - IO 1.sch		
Title: CIRCUIT PROTECTION SUBSYSTEM - I/O PIN 1		
Size: B	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 5/10

OVERVOLTAGE PROTECTION

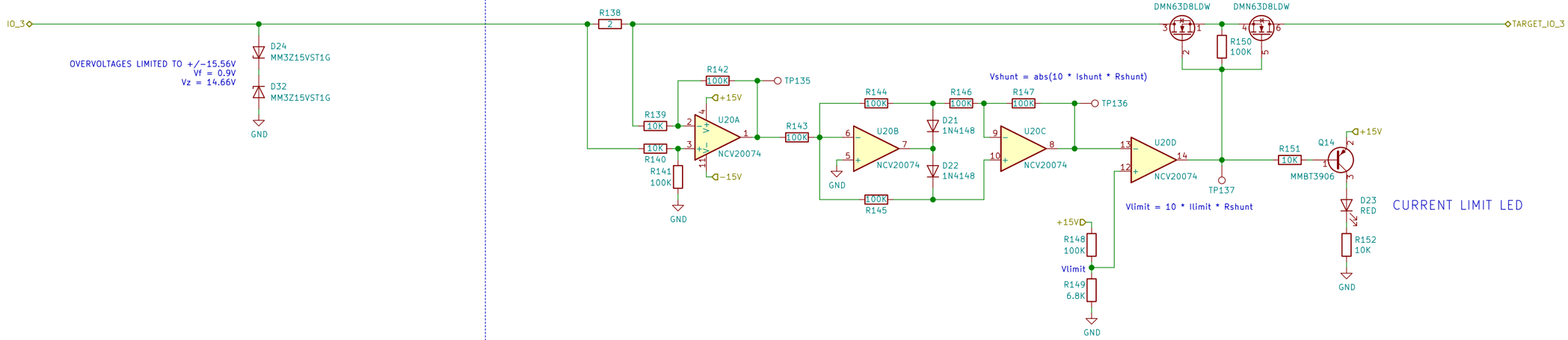
OVERCURRENT PROTECTION

NOTE: DECOUPLING CAPS,
PLACE CLOSE TO V+/V−



TRANSISTOR POWER DISSIPATION:
(WORST CASE: OUPUTTING +15V, PIN SHORTED TO −15V)
Pq = (15V − −15V) * 50mA = 1.5W

MAX AVG Pq IS 275mW.
ALLOWABLE DUTY CYCLE IS 0.275 / 1.5 = 0.18
(10 SEC SHORT CIRCUIT OVER 1 MIN)



CURRENT LIMIT LED

DT18 − I/O MASTER
THE UNIVERSITY OF AKRON

Sheet: /Circuit Protection − I/O Pin 3/		
File: Circuit Protection − IO 3.sch		
Title: CIRCUIT PROTECTION SUBSYSTEM − I/O PIN 3		
Size: B	Date: 2020−02−13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 6/10

OVERVOLTAGE PROTECTION

OVERCURRENT PROTECTION

NOTE: DECOUPLING CAPS,
PLACE CLOSE TO V+/V-

+15V
C115 0.1u
C116 0.1u
GND
-15V

TRANSISTOR POWER DISSIPATION:
(WORST CASE: OUPUTTING +15V, PIN SHORTED TO -15V)
Pq = (15V - -15V) * 50mA = 1.5W

MAX AVG Pq IS 275mW.
ALLOWABLE DUTY CYCLE IS 0.275 / 1.5 = 0.18
(10 SEC SHORT CIRCUIT OVER 1 MIN)

IO_4

OVERVOLTAGES LIMITED TO +/-15.56V
Vf = 0.9V
Vz = 14.66V

D36 MM3Z15VST1G
D37 MM3Z15VST1G
GND

R153 2

Q15A DMN63D8LDW
Q15B DMN63D8LDW
R165 100K

TARGET_IO_4

R157 100K
TP138
+15V
R154 10K
R155 10K
R156 100K
U21A NCV20074
-15V
GND
R158 100K

Vshunt = abs(10 * Ishunt * Rshunt)

R159 100K
R161 100K
R162 100K
TP139
U21B NCV20074
U21C NCV20074
D25 1N4148
D26 1N4148
R160 100K

U21D NCV20074
TP140
Vlimit = 10 * Ilimit * Rshunt

R166 10K
Q16 MMBT3906
+15V
D27 RED
R167 10K
GND

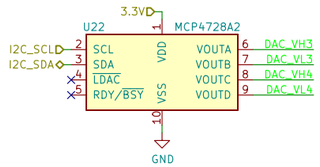
CURRENT LIMIT LED

+15V
R163 100K
Vlimit
R164 6.8K
GND

DT18 - I/O MASTER
THE UNIVERSITY OF AKRON

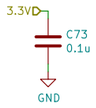
Sheet: /Circuit Protection - I/O Pin 4/		
File: Circuit Protection - IO 4.sch		
Title: CIRCUIT PROTECTION SUBSYSTEM - I/O PIN 4		
Size: B	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 7/10

LOGIC LEVEL GENERATOR

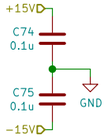


NOTE: P/N MCP4728A2 IS PRE-PROGRAMMED WITH I2C ADDRESS 0b1100010

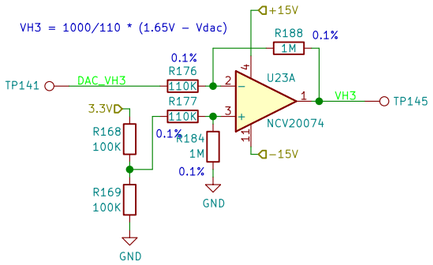
NOTE: DECOUPLING CAP, PLACE CLOSE TO VDD

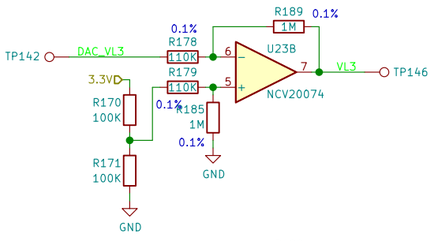


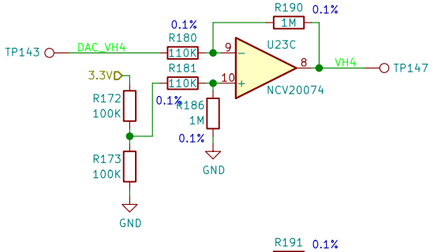
NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+/V-

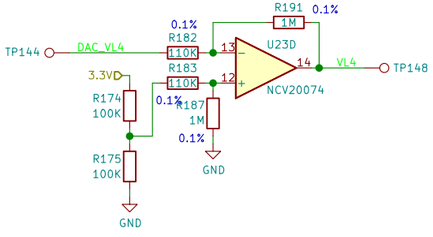


VH3 = 1000/110 * (1.65V - Vdac)



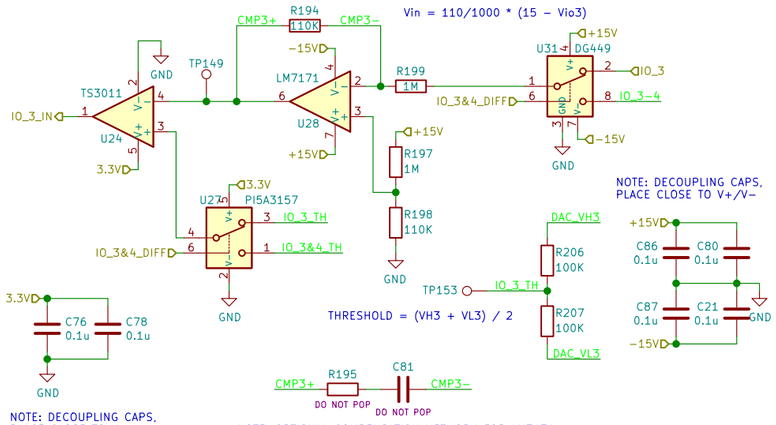






SINGLE-ENDED RECEIVER (I/O PIN 3)

Vin = 110/1000 * (15 - Vio3)



NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+/V-

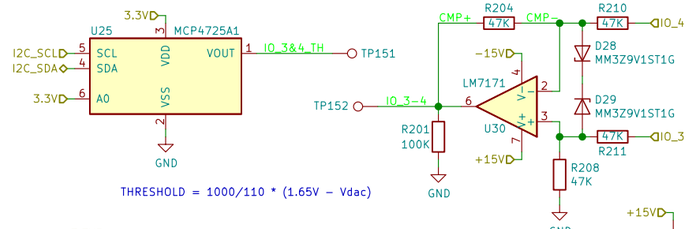
THRESHOLD = (VH3 + VL3) / 2

NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+

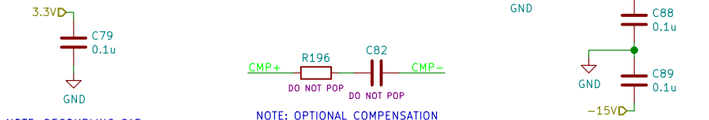
NOTE: OPTIONAL COMPENSATION NETWORK FOR LM7171

DIFFERENTIAL RECEIVER (I/O PIN 3 - I/O PIN 4)

NOTE: P/N MCP4725A1 IS PRE-PROGRAMMED WITH I2C ADDRESS 0b1100011



THRESHOLD = 1000/110 * (1.65V - Vdac)



NOTE: DECOUPLING CAP, PLACE CLOSE TO VDD

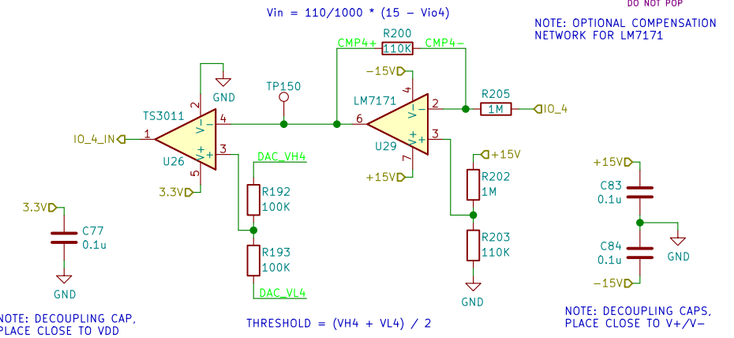
NOTE: OPTIONAL COMPENSATION NETWORK FOR LM7171

NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+/V-

SINGLE-ENDED RECEIVER (I/O PIN 4)

NOTE: OPTIONAL COMPENSATION NETWORK FOR LM7171

Vin = 110/1000 * (15 - Vio4)

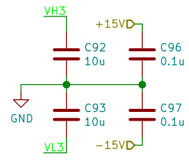


NOTE: DECOUPLING CAP, PLACE CLOSE TO VDD

THRESHOLD = (VH4 + VL4) / 2

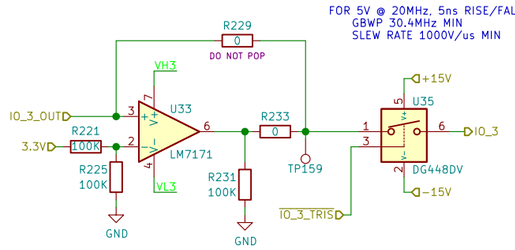
NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+/V-

OUTPUT DRIVER (I/O PIN 3)

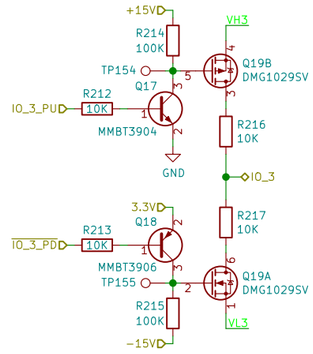


NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+/V-

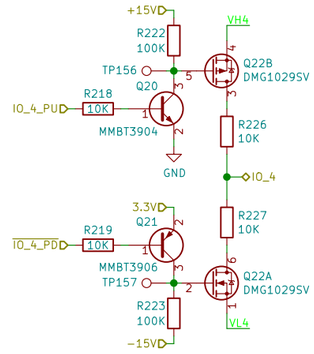
FOR 5V @ 20MHz, 5ns RISE/FALL:
GBWP 30.4MHz MIN
SLEW RATE 1000V/us MIN



CONFIGURABLE RESISTORS

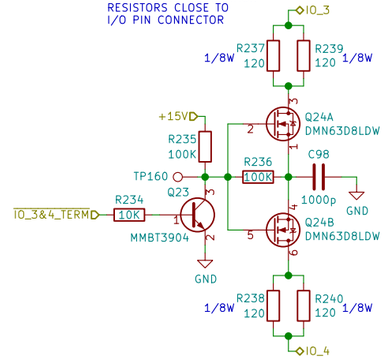


I/O PIN 3
10KΩ PULL-UP/DOWN



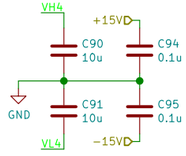
I/O PIN 4
10KΩ PULL-UP/DOWN

NOTE: PLACE TERMINATION RESISTORS CLOSE TO I/O PIN CONNECTOR



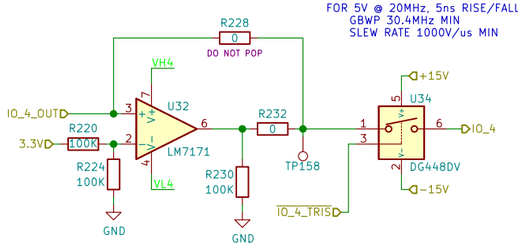
120Ω TERMINATION
BETWEEN I/O PINS 3 & 4

OUTPUT DRIVER (I/O PIN 4)



NOTE: DECOUPLING CAPS, PLACE CLOSE TO V+/V-

FOR 5V @ 20MHz, 5ns RISE/FALL:
GBWP 30.4MHz MIN
SLEW RATE 1000V/us MIN



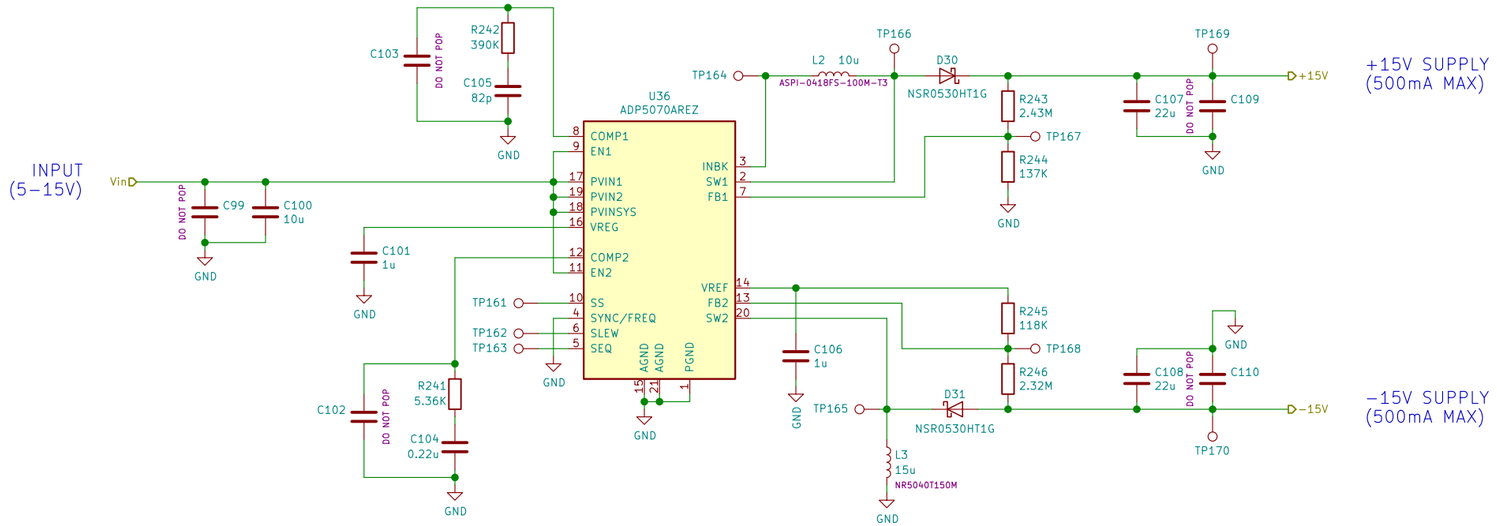
DT18 - I/O MASTER
THE UNIVERSITY OF AKRON

Sheet: /Level Shifter - I/O Pins 3&4/
File: Level Shifter - IO 3-4.sch

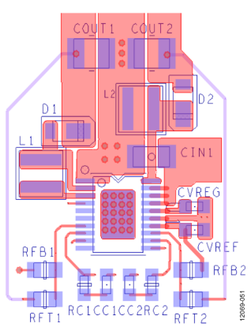
Title: LEVEL SHIFTER SUBSYSTEM - I/O PINS 3 & 4

Size: B	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 8/10

+/−15V REGULATOR



RECOMMENDED PCB LAYOUT



DT18 – I/O MASTER
THE UNIVERSITY OF AKRON
Sheet: /15V Regulator/
File: 15V Regulator.sch

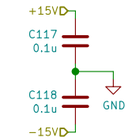
Title: +/−15V REGULATOR

Size: B	Date: 2020−02−13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 9/10

OVERVOLTAGE PROTECTION

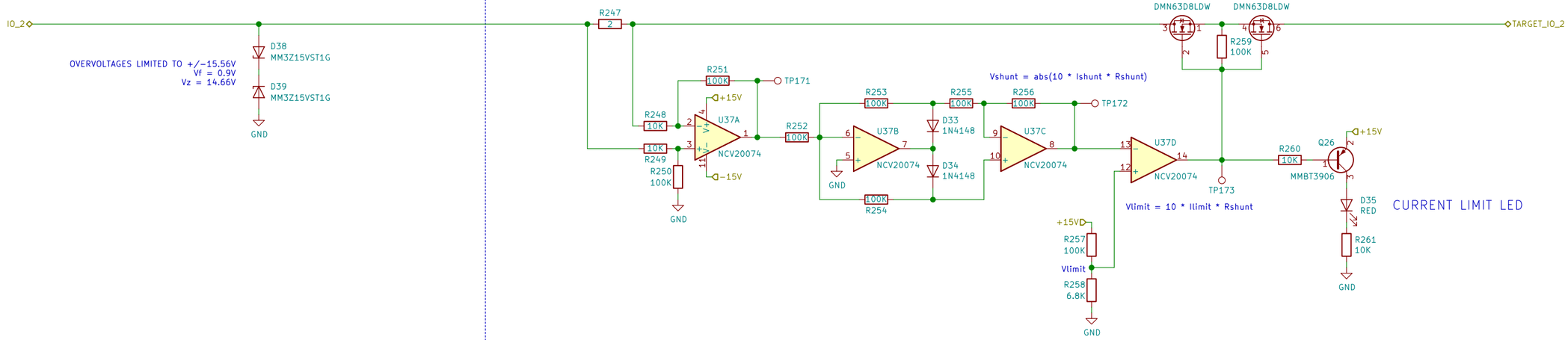
OVERCURRENT PROTECTION

NOTE: DECOUPLING CAPS,
PLACE CLOSE TO V+/V−



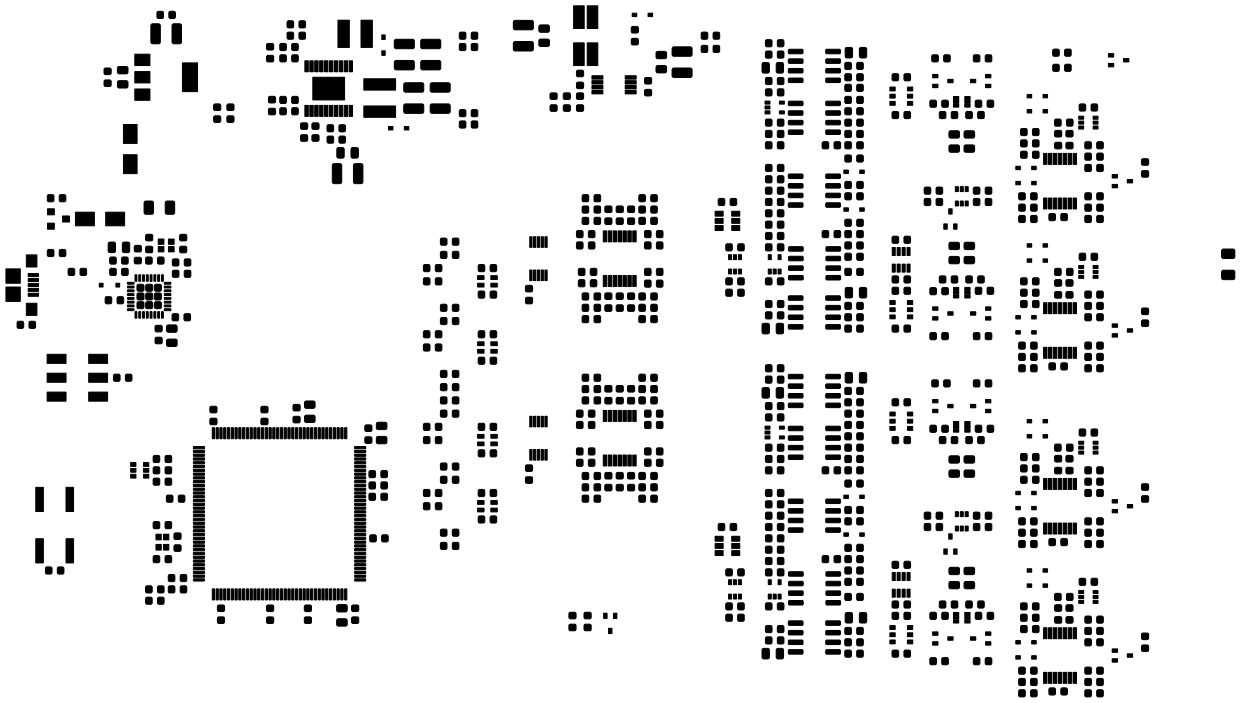
TRANSISTOR POWER DISSIPATION:
(WORST CASE: OUPUTTING +15V, PIN SHORTED TO −15V)
Pq = (15V − −15V) * 50mA = 1.5W

MAX AVG Pq IS 275mW.
ALLOWABLE DUTY CYCLE IS 0.275 / 1.5 = 0.18
(10 SEC SHORT CIRCUIT OVER 1 MIN)



DT18 − I/O MASTER		
THE UNIVERSITY OF AKRON		
Sheet: /Circuit Protection − I/O Pin 2/		
File: Circuit Protection − IO 2.sch		
Title: CIRCUIT PROTECTION SUBSYSTEM − I/O PIN 2		
Size: B	Date: 2020−02−13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 10/10

11.2 I/O Master v1.0 PCB Layout

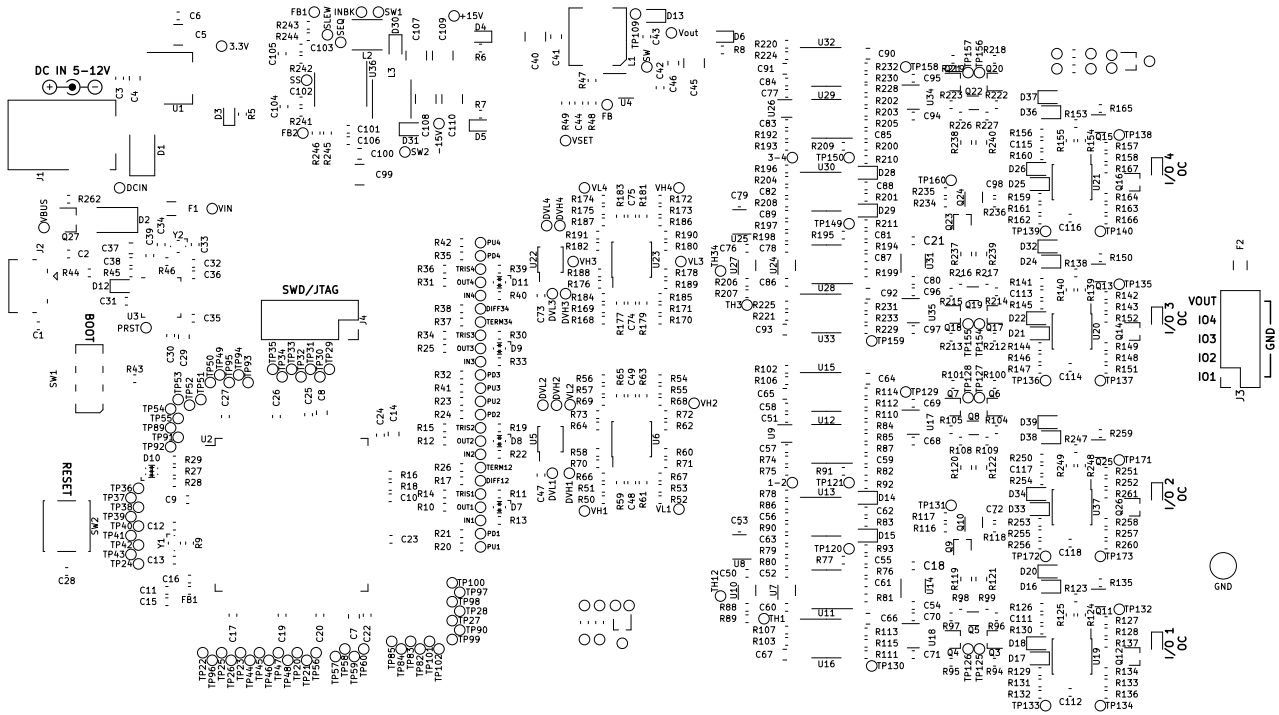


DT18 – I/O MASTER
THE UNIVERSITY OF AKRON

Sheet:
File: io-master.kicad_pcb

Title: I/O MASTER v1

Size: A	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1



DT18 – I/O MASTER

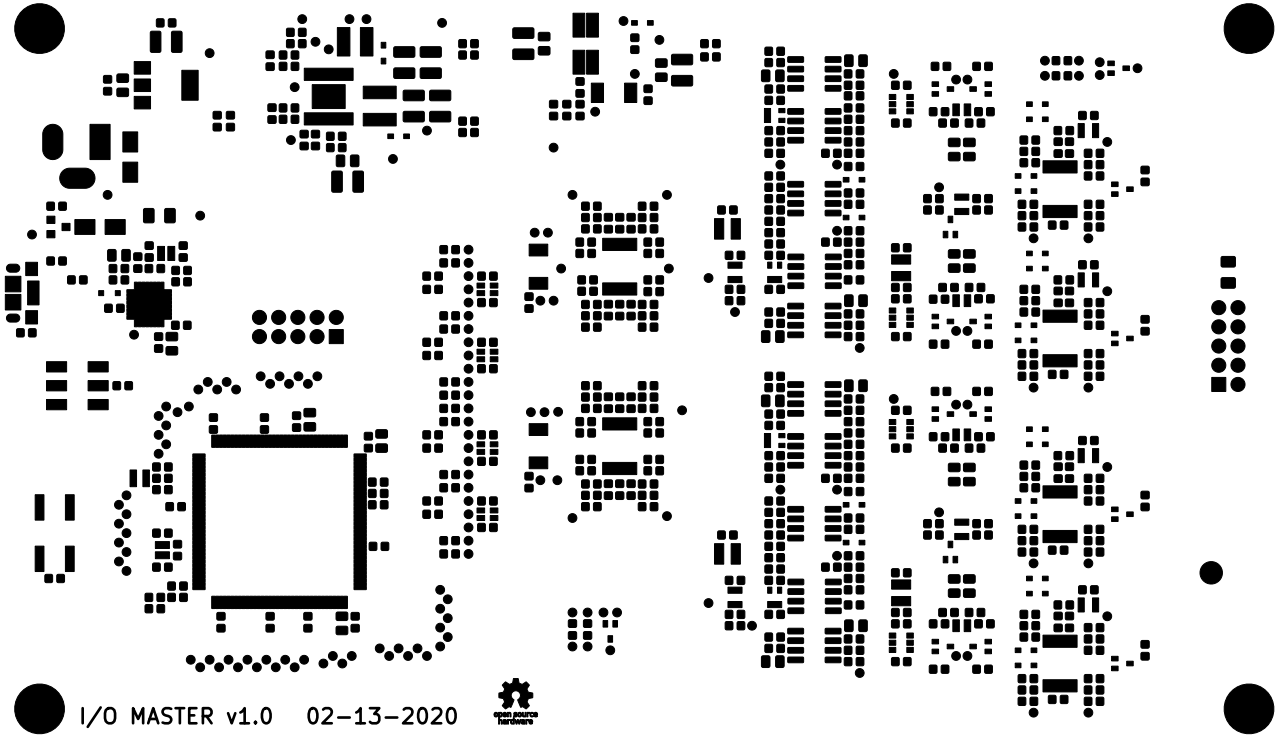
THE UNIVERSITY OF AKRON

Sheet:

File: io-master.kicad_pcb

Title: I/O MASTER v1

Size: A	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1

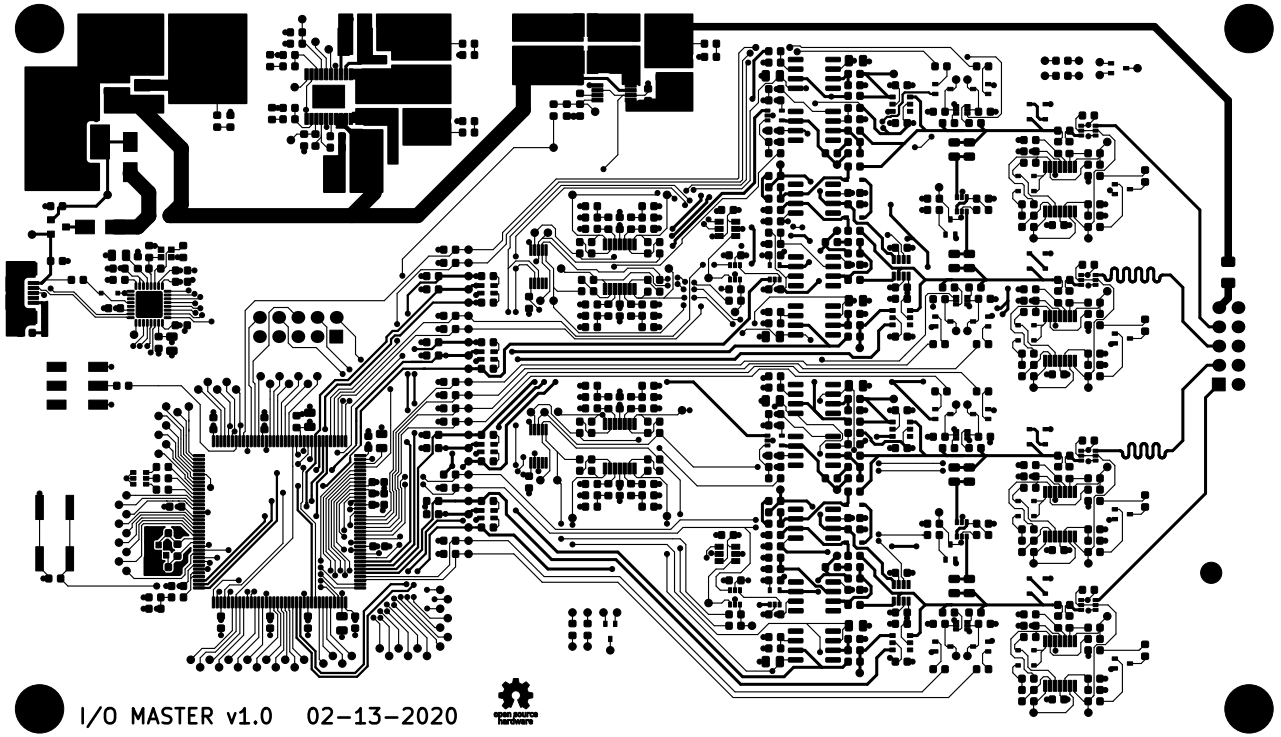


DT18 – I/O MASTER
THE UNIVERSITY OF AKRON

Sheet:
File: io-master.kicad_pcb

Title: I/O MASTER v1

Size: A	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1



DT18 - I/O MASTER
THE UNIVERSITY OF AKRON

Sheet:
File: io-master.kicad_pcb

Title: I/O MASTER v1

Size: A	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1

DT18 – I/O MASTER
THE UNIVERSITY OF AKRON

Sheet:
File: io-master.kicad_pcb

Title: I/O MASTER v1

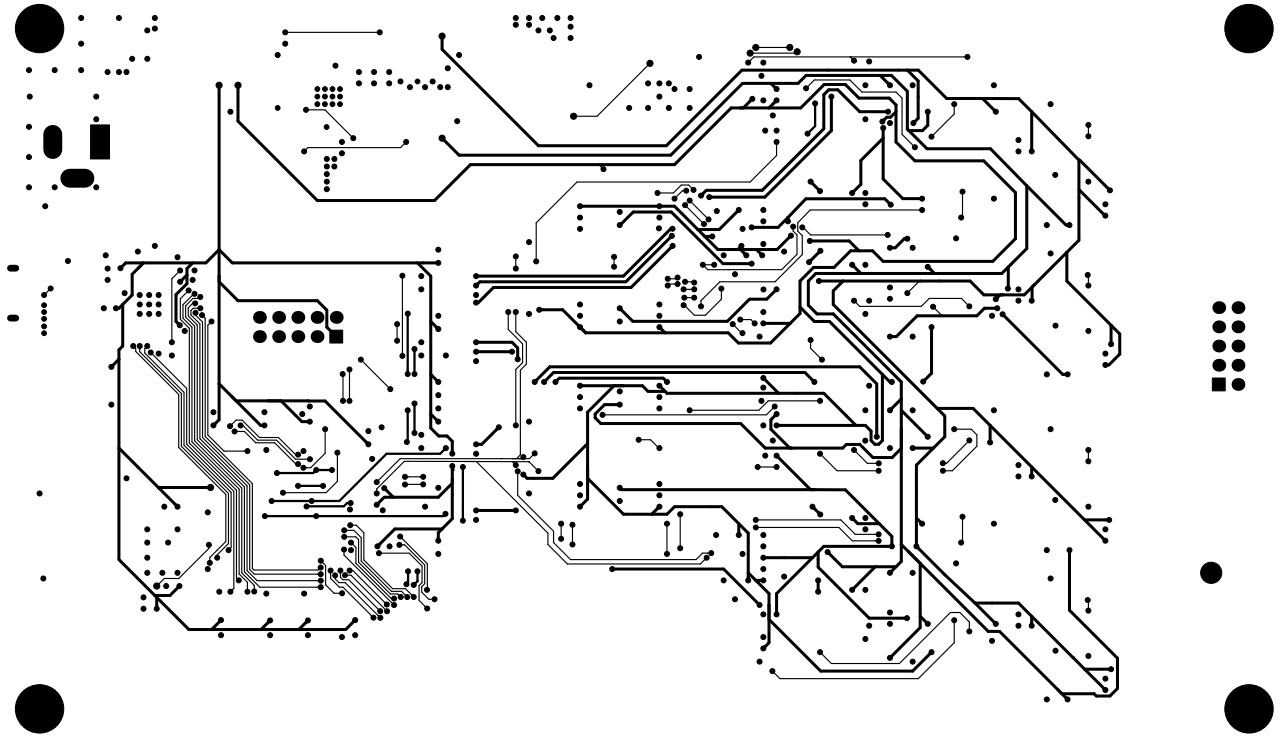
Size: A	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1

DT18 – I/O MASTER
THE UNIVERSITY OF AKRON

Sheet:
File: io-master.kicad_pcb

Title: I/O MASTER v1

Size: A	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1

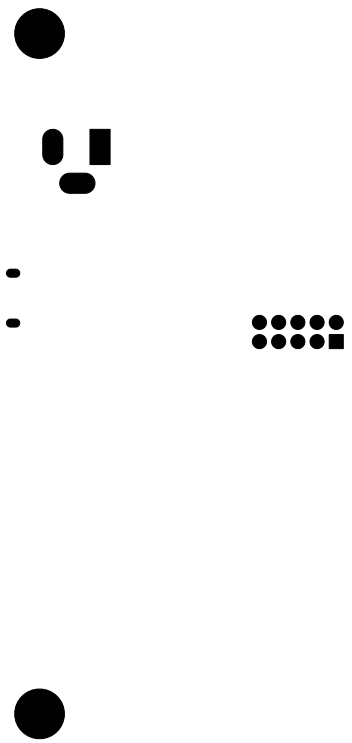


DT18 – I/O MASTER
THE UNIVERSITY OF AKRON

Sheet:
File: io-master.kicad_pcb

Title: I/O MASTER v1

Size: A	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1

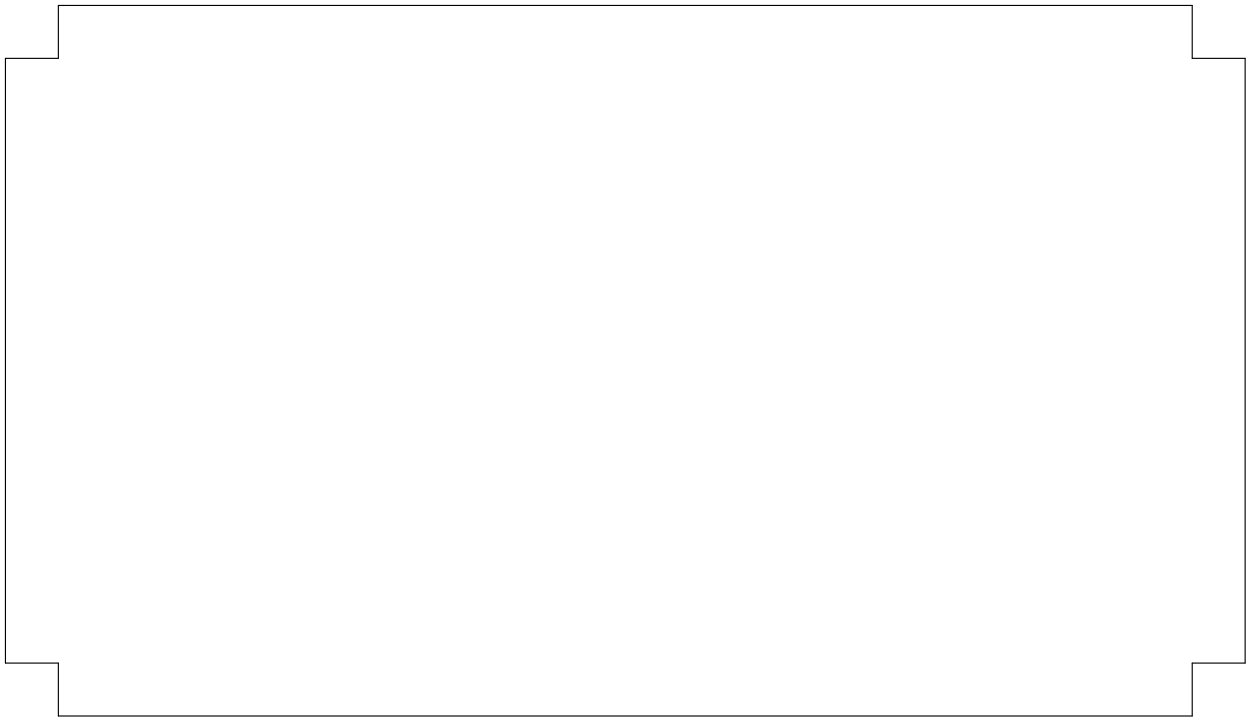


DT18 – I/O MASTER
THE UNIVERSITY OF AKRON

Sheet:
File: io-master.kicad_pcb

Title: I/O MASTER v1

Size: A	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1

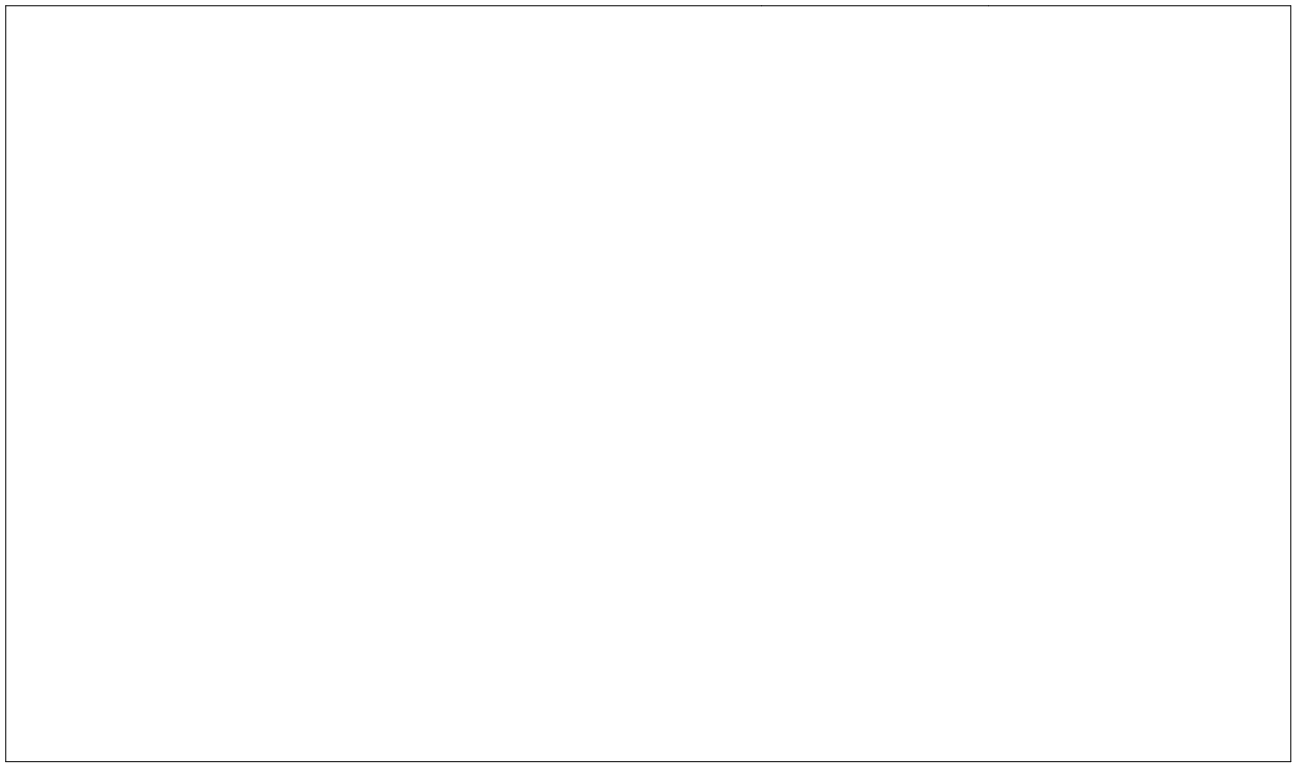


DT18 – I/O MASTER
THE UNIVERSITY OF AKRON

Sheet:
File: io-master.kicad_pcb

Title: I/O MASTER v1

Size: A	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1

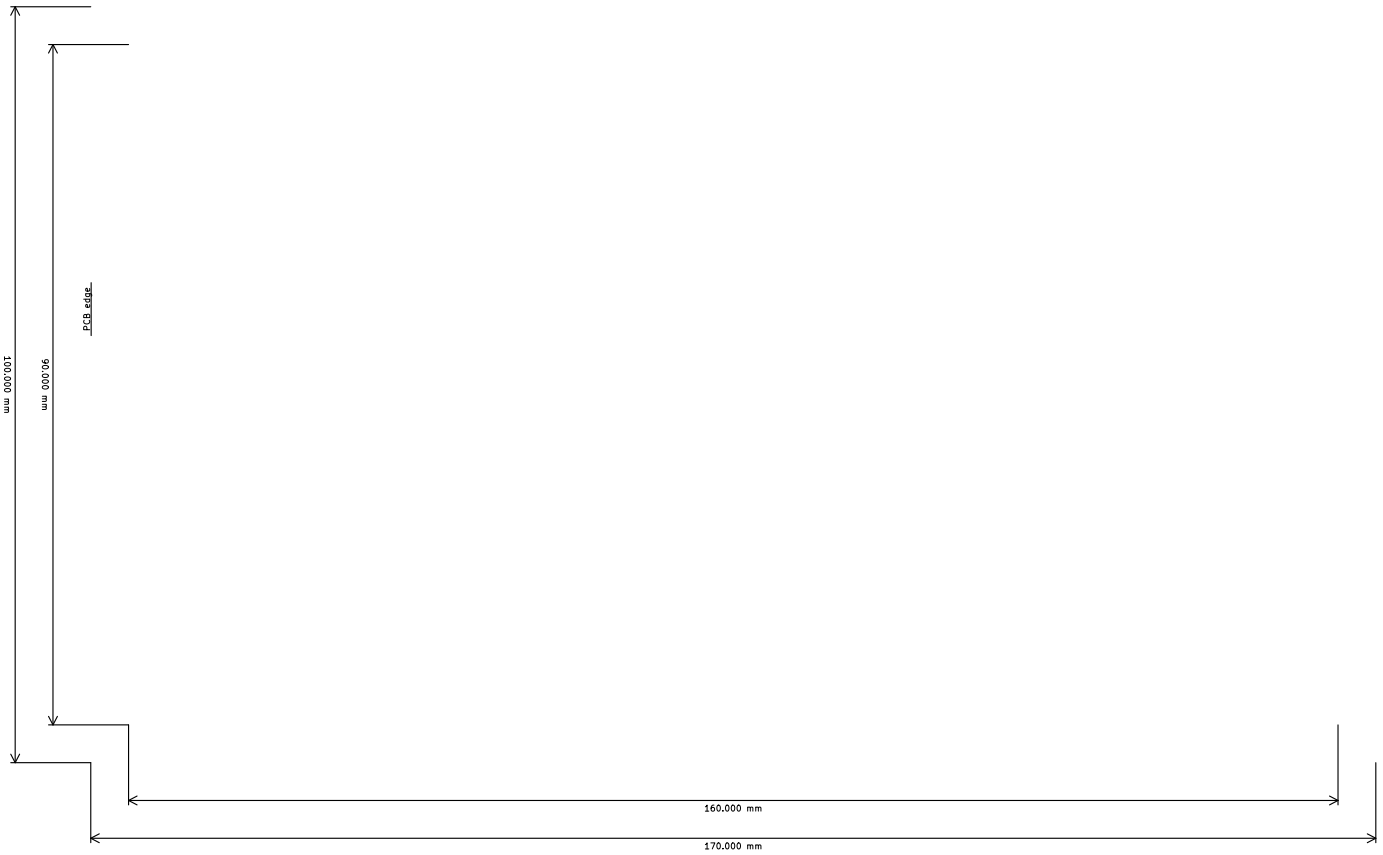


DT18 – I/O MASTER
THE UNIVERSITY OF AKRON

Sheet:
File: io-master.kicad_pcb

Title: I/O MASTER v1

Size: A	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1

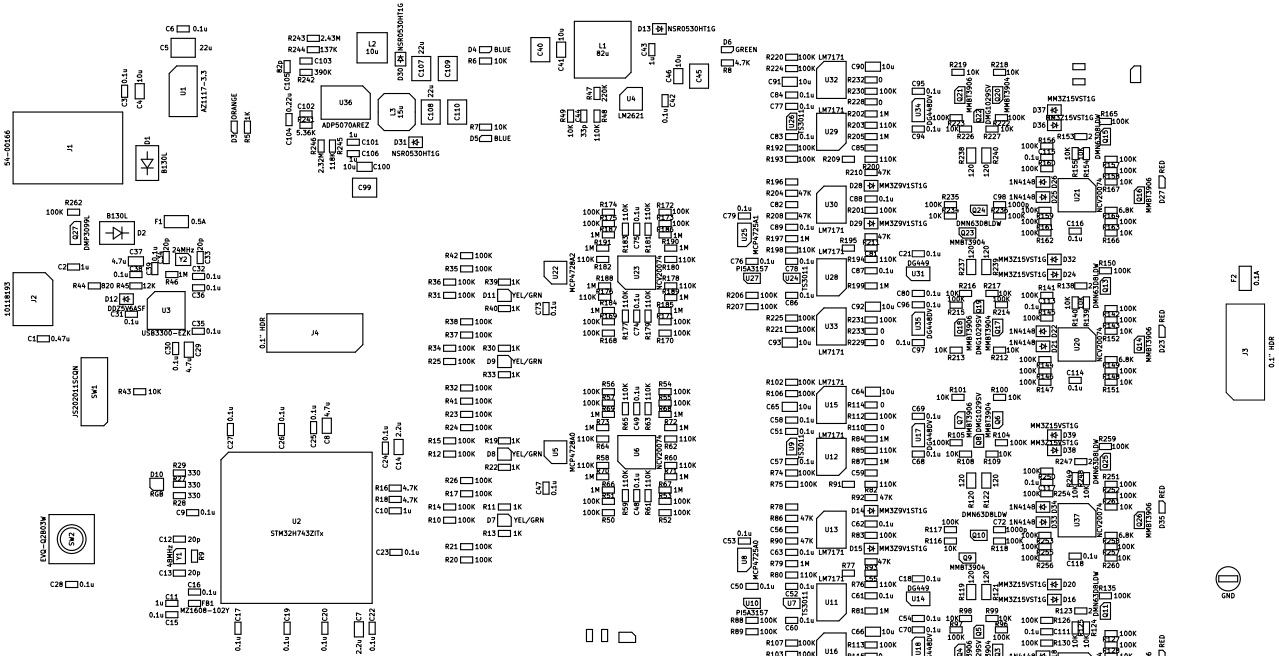


DT18 – I/O MASTER
THE UNIVERSITY OF AKRON

Sheet:
File: io-master.kicad_pcb

Title: I/O MASTER v1

Size: A	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1



DT18 – I/O MASTER		
THE UNIVERSITY OF AKRON		
Sheet:		
File: io-master.kicad_pcb		
Title: I/O MASTER v1		
Size: A	Date: 2020-02-13	**Rev: A**
KiCad E.D.A. kicad 5.1.5		Id: 1/1

11.3 I/O Master v1.0 BOM

I/O Master v1.0 - Populated Parts

						Qty 1		Qty 100	
Qty.	**Refdes.**	**Description**	**Manufacturer**	**Manufacturer PN**	**Digikey PN**	**Unit Price**	**Extended Price**	**Unit Price**	**Extended Price**
1	PCB1	Bare PCB, 4 layer, 1oz copper, 1.6mm, ENIG, 100x100mm w/ stencil	JLCPCB			$11.35	$11.35	$1.81	$1.81
1	C1	0.47uF Ceramic Capacitor, 10%, 25V, X7R, 0603	Samsung	CL10B474KA8NFNC	1276-2082-1-ND	$0.12	$0.12	$0.04	$0.04
1	C2	1uF Ceramic Capacitor, 10%, 25V, X5R, 0603	Samsung	CL10A105KA8NNNC	1276-1102-1-ND	$0.10	$0.10	$0.02	$0.02
2	D1, D2	Schottky diode, Vf=0.41V, 1A, SMA	Diodes Inc	B130L-13-F	B130L-FDICT-ND	$0.36	$0.72	$0.17	$0.33
1	D3	Orange LED, Vf=2V, 0603	Lite-On Inc.	LTST-C190KFKT	160-1434-1-ND	$0.24	$0.24	$0.08	$0.08
2	D4, D5	Blue LED, Vf=3.2V, 0603	Wurth	150060BS55040	732-12013-1-ND	$0.18	$0.36	$0.15	$0.29
1	D6	Green LED, Vf=2V, 0603	Würth Elektronik	150060VS75000	732-4980-1-ND	$0.14	$0.14	$0.11	$0.11
1	F1	PTC Fuse, 500mA hold, 1A trip, 1206	Bel Fuse	0ZCJ0050FF2G	507-1802-1-ND	$0.13	$0.13	$0.10	$0.10
1	F2	PTC Fuse, 100mA hold, 250mA trip, 1206	Bel Fuse	0ZCJ0010FF2E	507-1794-1-ND	$0.13	$0.13	$0.10	$0.10
1	J1	DC Barrel Jack, 5.5mm OD, 2.1mm ID, THT	Tensility	54-00166	839-54-00166-ND	$0.47	$0.47	$0.33	$0.33
1	J2	Micro USB Connector, SMD	Amphenol	10118193-0001LF	609-4616-1-ND	$0.43	$0.43	$0.31	$0.31
1	J3	0.1" Header, 2x5 pins, THT	Sullins Connector	PPPC052LFBN-RC	S7108-ND	$0.71	$0.71	$0.54	$0.54
1	Q27	P-ch MOSFET, 30V, 3.8A, SOT-23-3	Diodes Inc	DMP3099L-7	DMP3099L-7DICT-ND	$0.34	$0.34	$0.12	$0.12
1	R5	1Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW06031K00FKEA	541-1.00KHCT-ND	$0.10	$0.10	$0.01	$0.01
2	R6, R7	10Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW060310K0FKEA	541-10.0KHCT-ND	$0.10	$0.20	$0.01	$0.03
1	R8	4.7Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW06034K70FKEA	541-4.70KHCT-ND	$0.10	$0.10	$0.01	$0.01
1	R262	100Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW0603100KFKEAC	541-3950-1-ND	$0.10	$0.10	$0.01	$0.01
1	TP4	Multipurpose Test Point	Keystone	5011	36-5011-ND	$0.35	$0.35	$0.30	$0.30
3.3V Regulator									
1	U1	Linear Regulator, 4.6-15V input, 3.3V output, 800mA, SOT-223	Diodes Inc	AZ1117IH-3.3TRG1	AZ1117IH-3.3TRG1DICT-ND	$0.37	$0.37	$0.18	$0.18
2	C3, C6	0.1uF Ceramic Capacitor, 10%, 25V, X7R, 0603	Samsung	CL10B104KA8NNNC	1276-1006-1-ND	$0.10	$0.20	$0.01	$0.02
1	C4	10uF Ceramic Capacitor, 10%, 25V, X5R, 0805	Samsung	CL21A106KAYNNNG	1276-6454-1-ND	$0.25	$0.25	$0.09	$0.09
1	C5	22uF Ceramic Capacitor, 10%, 25V, X5R, 1210	Samsung	CL32A226KAJNNNE	1276-1101-1-ND	$0.57	$0.57	$0.26	$0.26
+/-15V Regulator									
1	C100	10uF Ceramic Capacitor, 10%, 25V, X5R, 0805	Samsung	CL21A106KAYNNNG	1276-6454-1-ND	$0.25	$0.25	$0.09	$0.09
2	C101, C106	1uF Ceramic Capacitor, 10%, 25V, X5R, 0603	Samsung	CL10A105KA8NNNC	1276-1102-1-ND	$0.10	$0.20	$0.02	$0.04
1	C104	0.22uF Ceramic Capacitor, 10%, 25V, X5R, 0603	Samsung	CL10A224KA8NNNC	1276-1875-1-ND	$0.12	$0.12	$0.04	$0.04
1	C105	82pF Ceramic Capacitor, 5%, 50V, C0G, 0603	Samsung	CL10C820JB8NNNC	1276-1242-1-ND	$0.10	$0.10	$0.03	$0.03
2	C107, C108	22uF Ceramic Capacitor, 10%, 25V, X5R, 1210	Samsung	CL32A226KAJNNNE	1276-1101-1-ND	$0.57	$1.14	$0.26	$0.53
2	D30, D31	Schottky Diode, Vf=0.62V, 30V, 500mA, SOD-323	ON Semi	NSR0530HT1G	NSR0530HT1GOSCT-ND	$0.16	$0.32	$0.06	$0.12
1	L2	10uH Inductor, 1.3A rated, 1.4A saturation, shielded, SMD	Abracon	ASPI-0418FS-100M-T3	535-10763-1-ND	$0.55	$0.55	$0.35	$0.35
1	L3	15uH Inductor, 1.8A rated, 2.0A saturation, shielded, SMD	Taiyo Yuden	NR5040T150M	587-2366-1-ND	$0.45	$0.45	$0.28	$0.28
1	R241	5.36Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW06035K36FKEA	541-5.36KHCT-ND	$0.10	$0.10	$0.01	$0.01
1	R242	390Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW0603390KFKEA	541-390KHCT-ND	$0.10	$0.10	$0.01	$0.01
1	R243	2.43Mohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW06032M43FKEA	541-2.43MHCT-ND	$0.10	$0.10	$0.01	$0.01
1	R244	137Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW0603137KFKEA	541-137KHCT-ND	$0.10	$0.10	$0.01	$0.01
1	R245	118Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW0603118KFKEA	541-118KHCT-ND	$0.10	$0.10	$0.01	$0.01
1	R246	2.32Mohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW06032K32FKEA	541-2.32KHCT-ND	$0.10	$0.10	$0.01	$0.01
1	U36	Switching Regulator, pos & neg output, 2.5-15V input, 20-TSSOP	Analog Devices	ADP5070AREZ	ADP5070AREZ-R7CT-ND	$6.10	$6.10	$4.57	$4.57
3.3-15V Adjustable Regulator									
2	C41, C46	10uF Ceramic Capacitor, 10%, 25V, X5R, 0805	Samsung	CL21A106KAYNNNG	1276-6454-1-ND	$0.25	$0.50	$0.09	$0.17
1	C42	0.1uF Ceramic Capacitor, 10%, 25V, X7R, 0603	Samsung	CL10B104KA8NNNC	1276-1006-1-ND	$0.10	$0.10	$0.01	$0.01
1	C43	1uF Ceramic Capacitor, 10%, 25V, X5R, 0603	Samsung	CL10A105KA8NNNC	1276-1102-1-ND	$0.10	$0.10	$0.02	$0.02
1	C44	33pF Ceramic Capacitor, 5%, 50V, C0G, 0603	Samsung	CL10C330JB8NNNC	1276-1070-1-ND	$0.10	$0.10	$0.02	$0.02
1	D13	Schottky Diode, Vf=0.62V, 30V, 500mA, SOD-323	ON Semi	NSR0530HT1G	NSR0530HT1GOSCT-ND	$0.16	$0.16	$0.06	$0.06
1	L1	82uH Inductor, dual windings, 0.86A rated, 0.89A saturation, shielded, SMD	Bourns	SRF0703-820M	SRF0703-820MCT-ND	$1.01	$1.01	$0.68	$0.68
1	R47	220Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW0603220KFKEAC	541-4039-1-ND	$0.10	$0.10	$0.01	$0.01
1	R48	110Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW0603110KFKEA	541-110KHCT-ND	$0.10	$0.10	$0.01	$0.01
1	R49	10Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW060310K0FKEA	541-10.0KHCT-ND	$0.10	$0.10	$0.01	$0.01
1	U4	Switching Regulator, 1.2-14V input, 1.24-14V output, 2.85A, 8-TSSOP	Texas Instruments	LM2621MMX/NOPB	LM2621MMX/NOPBCT-ND	$2.01	$2.01	$1.36	$1.36
Microcontroller Subsystem									
2	C7, C14	2.2uF Ceramic Capacitor, 10%, 25V, X5R, 0805	Samsung	CL21A225KAFNNNG	1276-6458-1-ND	$0.16	$0.32	$0.06	$0.11
3	C8, C29, C37	4.7uF Ceramic Capacitor, 10%, 25V, X5R, 0805	Samsung	CL21A475KAQNNNG	1276-6462-1-ND	$0.17	$0.51	$0.06	$0.17

I/O Master v1.0 - Populated Parts

						Qty 1		Qty 100	
Qty.	**Refdes.**	**Description**	**Manufacturer**	**Manufacturer PN**	**Digikey PN**	**Unit Price**	**Extended Price**	**Unit Price**	**Extended Price**
20	C9, C15, C16, C17, C19, C20, C22, C23, C24, C25, C26, C27, C28, C30, C31, C32, C35, C36, C38, C39	0.1uF Ceramic Capacitor, 10%, 25V, X7R, 0603	Samsung	CL10B104KA8NNNC	1276-1006-1-ND	$0.10	$2.00	$0.01	$0.25
2	C10, C11	1uF Ceramic Capacitor, 10%, 25V, X5R, 0603	Samsung	CL10A105KA8NNNC	1276-1102-1-ND	$0.10	$0.20	$0.02	$0.04
4	C12, C13, C33, C34	20pF Ceramic Capacitor, 5%, 50V, C0G, 0603	Samsung	CL10C200JB8NNNC	1276-1187-1-ND	$0.10	$0.40	$0.02	$0.10
4	D7, D8, D9, D11	Green/Yellow LED, independent, SMD	Kingbright	APTB1615YSGC-F01	754-1151-1-ND	$0.42	$1.68	$0.20	$0.79
1	D10	RGB LED, independent, SMD	Everlight Electronics	19-337C/RSBHGHC-A88/4T	1080-1631-1-ND	$0.44	$0.44	$0.21	$0.21
1	D12	Zener Diode, Vz=5.42V, SOD-323F	Diodes Inc	DDZ5V6ASF-7	DDZ5V6ASF-7DICT-ND	$0.16	$0.16	$0.06	$0.06
1	FB1	Ferrite Bead, 100mA, 0603	Bourns	MZ1608-102Y	MZ1608-102YCT-ND	$0.10	$0.10	$0.04	$0.04
1	J4	0.1" Header, 2x5 pins, THT	Sullins Connector	PPPC052LFBN-RC	S7108-ND	$0.71	$0.71	$0.54	$0.54
20	R10, R12, R14, R15, R17, R20, R21, R23, R24, R25, R26, R31, R32, R34, R35, R36, R37, R38, R41, R42	100Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW0603100KFKEAC	541-3950-1-ND	$0.10	$2.00	$0.01	$0.28
8	R11, R13, R19, R22, R30, R33, R39, R40	1Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW06031K00FKEA	541-1.00KHCT-ND	$0.10	$0.80	$0.01	$0.11
2	R16, R18	4.7Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW06034K70FKEA	541-4.70KHCT-ND	$0.10	$0.20	$0.01	$0.03
3	R27, R28, R29	330ohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW0603330RFKEA	541-330HCT-ND	$0.10	$0.30	$0.01	$0.04
1	R43	10Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW060310K0FKEA	541-10.0KHCT-ND	$0.10	$0.10	$0.01	$0.01
1	R44	820ohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW0603820RFKEA	541-820HCT-ND	$0.10	$0.10	$0.01	$0.01
1	R45	12Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW060312K0FKEAC	541-4046-1-ND	$0.10	$0.10	$0.01	$0.01
1	R46	1Mohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW06031M00FKEA	541-1.00MHCT-ND	$0.10	$0.10	$0.01	$0.01
1	SW1	Slide Switch, SPDT, SMD	C&K	JS202011SCQN	401-2002-1-ND	$0.48	$0.48	$0.39	$0.39
1	SW2	Push Button, SPST-NO, SMD	Panasonic	EVQ-Q2B03W	P12932SCT-ND	$0.12	$0.12	$0.12	$0.12
1	U2	32-bit Microcontroller, 144-LQFP	STmicro	STM32H743ZIT6	497-17721-ND	$10.65	$10.65	$8.27	$8.27
1	U3	USB 2.0 HS PHY, 32-QFN	Microchip	USB3300-EZK-TR	USB3300-EZK-CT-ND	$1.24	$1.24	$0.99	$0.99
1	Y1	Crystal, 48MHz, SMD	Murata Electronics	XRCGB48M000F4M02R0	490-18354-1-ND	$0.35	$0.35	$0.24	$0.24
1	Y2	Crystal, 24MHz, SMD	Murata Electronics	XRCGB24M000F0L00R0	490-5575-1-ND	$0.35	$0.35	$0.24	$0.24
Level Shifter Subsystem									
38	C18, C21, C47, C48, C49, C50, C51, C52, C53, C54, C57, C58, C60, C61, C62, C63, C68, C69, C70, C71, C73, C74, C75, C76, C77, C78, C79, C80, C83, C84, C86, C87, C88, C89, C94, C95, C96, C97	0.1uF Ceramic Capacitor, 10%, 25V, X7R, 0603	Samsung	CL10B104KA8NNNC	1276-1006-1-ND	$0.10	$3.80	$0.01	$0.47
8	C64, C65, C66, C67, C90, C91, C92, C93	10uF Ceramic Capacitor, 10%, 25V, X5R, 0805	Samsung	CL21A106KAYNNNG	1276-6454-1-ND	$0.25	$2.00	$0.09	$0.68
2	C72, C98	1000pF Ceramic Capacitor, 10%, 50V, X7R, 0603	Samsung	CL10B102KB8WPNC	1276-6838-1-ND	$0.10	$0.20	$0.01	$0.03
4	D14, D15, D28, D29	Zener Diode, Vz=9.1V, SOD-323	ON Semi	MM3Z9V1ST1G	MM3Z9V1ST1GOSCT-ND	$0.13	$0.52		$0.00
6	Q3, Q6, Q9, Q17, Q20, Q23	NPN Transistor, 40V, 200mA, SOT-323	ON Semi	MMBT3904WT1G	MMBT3904WT1GOSCT-ND	$0.13	$0.78	$0.05	$0.31
4	Q4, Q7, Q18, Q21	PNP Transistor, 40V, 200mA, SOT-323	ON Semi	SMMBT3906WT1G	SMMBT3906WT1GOSCT-ND	$0.12	$0.48	$0.05	$0.19
4	Q5, Q8, Q19, Q22	P-ch N-ch MOSFET Pair, 60V, 360mA, SOT-563	Diodes Inc	DMG1029SV-7	DMG1029SV-7DICT-ND	$0.39	$1.56	$0.18	$0.73
2	Q10, Q24	Dual N-ch MOSFET Pair, 30V 220mA, SOT-363	ON Semi	DMN63D8LDW-7	DMN63D8LDW-7CT-ND	$0.31	$0.62	$0.10	$0.21
50	R50, R51, R52, R53, R54, R55, R56, R57, R74, R75, R83, R88, R89, R96, R97, R102, R103, R104, R105, R106, R107, R112, R113, R117, R118, R168, R169, R170, R171, R172, R173, R174, R175, R192, R193, R201, R206, R207, R214, R215, R220, R221, R222, R223, R224, R225, R230, R231, R235, R236	100Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW0603100KFKEAC	541-3950-1-ND	$0.10	$5.00	$0.01	$0.70
16	R58, R59, R60, R61, R62, R63, R64, R65, R176, R177, R178, R179, R180, R181, R182, R183	110Kohm Resistor, 0.1%, 1/10W, 0603	Panasonic	ERA-3AEB114V	P110KDBCT-ND	$0.35	$5.60	$0.12	$1.88
16	R66, R67, R68, R69, R70, R71, R72, R73, R184, R185, R186, R187, R188, R189, R190, R191	1Mohm Resistor, 0.1%, 1/10W, 0603	Yageo	RT0603BRD071ML	YAG4498CT-ND	$0.33	$5.28	$0.11	$1.76
8	R76, R80, R82, R85, R194, R198, R200, R203	110Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW0603110KFKEA	541-110KHCT-ND	$0.10	$0.80	$0.01	$0.11
8	R79, R81, R84, R87, R197, R199, R202, R205	1Mohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW06031M00FKEA	541-1.00MHCT-ND	$0.10	$0.80	$0.01	$0.11
8	R86, R90, R92, R93, R204, R208, R210, R211	47Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW060347K0FKEAC	541-4021-1-ND	$0.10	$0.80	$0.01	$0.11
18	R94, R95, R98, R99, R100, R101, R108, R109, R116, R212, R213, R216, R217, R218, R219, R226, R227, R234	10Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW060310K0FKEA	541-10.0KHCT-ND	$0.10	$1.80	$0.01	$0.25
4	R114, R115, R232, R233	Jumper, 1/10W, 0603	Yageo	RC0603JR-070RL	311-0.0GRCT-ND	$0.10	$0.40	$0.01	$0.03
8	R119, R120, R121, R122, R237, R238, R239, R240	120ohm Resistor, 1%, 1/8W, 0805	Yageo	RC0805FR-07120RL	311-120CRCT-ND	$0.10	$0.80	$0.02	$0.13
1	U5	DAC, 12-bit, 4 channel, address = 000, 10-MSOP	Microchip	MCP4728A0T-E/UN	MCP4728A0T-E/UNCT-ND	$1.89	$1.89	$1.44	$1.44
2	U6, U23	Quad Op-Amp, rail-to-rail, +/-15V supply, 14-TSSOP	ON Semi	NCV20074DTBR2G	NCV20074DTBR2GOSCT-ND	$1.01	$2.02	$0.66	$1.31

I/O Master v1.0 - Populated Parts						Qty 1		Qty 100	
Qty.	**Refdes.**	**Description**	**Manufacturer**	**Manufacturer PN**	**Digikey PN**	**Unit Price**	**Extended Price**	**Unit Price**	**Extended Price**
4	U7, U9, U24, U26	Comparator, high speed, 2.2-5V supply, SOT-353	STmicro	TS3011ICT	497-11990-1-ND	$1.01	$4.04	$0.66	$2.62
1	U8	DAC, 12-bit, 1 channel, address = 00, SOT-23-6	Microchip	MCP4725A0T-E/CH	MCP4725A0T-E/CHCT-ND	$0.96	$0.96	$0.73	$0.73
2	U10, U27	Analog Switch, 1.65-5.5V supply, SPDT, SOT-363	Diodes Inc	PI5A3157CEX	PI5A3157CEXCT-ND	$0.36	$0.72	$0.18	$0.36
10	U11, U12, U13, U15, U16, U28, U29, U30, U32, U33	Op-Amp, high speed, +/-15V supply, 8-SOIC	Texas Instruments	LM7171BIMX/NOPB	LM7171BIMX/NOPBCT-ND	$2.89	$28.90	$2.02	$20.20
2	U14, U31	Analog Switch, +/-15V supply, SOT-23-8	Diodes Inc	DG449DS-T1-E3	DG449DS-T1-E3CT-ND	$1.50	$3.00	$0.98	$1.96
4	U17, U18, U34, U35	Analog Switch, high speed, +/-15V supply, SPST-NO, 6-TSOP	Vishay	DG448DV-T1-E3	DG448DV-T1-E3CT-ND	$1.29	$5.16	$0.84	$3.37
1	U22	DAC, 12-bit, 4 channel, address = 010, 10-MSOP	Microchip	MCP4728A2-E/UN	only available on MicrochipDirect	$1.89	$1.89	$1.44	$1.44
1	U25	DAC, 12-bit, 1 channel, address = 01, SOT-23-6	Microchip	MCP4725A1T-E/CH	MCP4725A1T-E/CHCT-ND	$0.96	$0.96	$0.73	$0.73
Circuit Protection Subsystem									
8	C111, C112, C113, C114, C115, C116, C117, C118	0.1uF Ceramic Capacitor, 10%, 25V, X7R, 0603	Samsung	CL10B104KA8NNNC	1276-1006-1-ND	$0.10	$0.80	$0.01	$0.10
8	D16, D20, D24, D32, D36, D37, D38, D39	Zener Diode, Vz=14.66V, SOD-323	ON Semi	MM3Z15VST1G	MM3Z15VST1GOSCT-ND	$0.13	$1.04	$0.05	$0.41
8	D17, D18, D21, D22, D25, D26, D33, D34	Diode, SOD-323	Diodes Inc	1N4148WS-7-F	1N4148WS-FDICT-ND	$0.17	$1.36	$0.07	$0.53
4	D19, D23, D27, D35	Red LED, Vf=1.9V, 0603	Würth Elektronik	150060SS75000	732-4979-1-ND	$0.14	$0.56	$0.11	$0.46
4	Q11, Q13, Q15, Q25	Dual N-ch MOSFET Pair, 30V 220mA, SOT-363	ON Semi	DMN63D8LDW-7	DMN63D8LDW-7CT-ND	$0.31	$1.24	$0.10	$0.41
4	Q12, Q14, Q16, Q26	PNP Transistor, 40V, 200mA, SOT-323	ON Semi	SMMBT3906WT1G	SMMBT3906WT1GOSCT-ND	$0.12	$0.48	$0.05	$0.19
4	R123, R138, R153, R247	2ohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW06032R00FKEA	541-2.00HHCT-ND	$0.10	$0.40	$0.02	$0.07
16	R124, R125, R136, R137, R139, R140, R151, R152, R154, R155, R166, R167, R248, R249, R260, R261	10Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW060310K0FKEA	541-10.0KHCT-ND	$0.10	$1.60	$0.01	$0.22
36	R126, R127, R128, R129, R130, R131, R132, R133, R135, R141, R142, R143, R144, R145, R146, R147, R148, R150, R156, R157, R158, R159, R160, R161, R162, R163, R165, R250, R251, R252, R253, R254, R255, R256, R257, R259	100Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW0603100KFKEAC	541-3950-1-ND	$0.10	$3.60	$0.01	$0.50
4	R134, R149, R164, R258	6.8Kohm Resistor, 1%, 1/10W, 0603	Vishay Dale	CRCW06036K80FKEA	541-6.80KHCT-ND	$0.10	$0.40	$0.01	$0.06
4	U19, U20, U21, U37	Quad Op-Amp, rail-to-rail, +/-15V supply, 14-TSSOP	ON Semi	NCV20074DTBR2G	NCV20074DTBR2GOSCT-ND	$1.01	$4.04	$0.66	$2.63
						BOM Total	**$151.30**	**BOM Total**	**$74.07**

I/O Master v1.0 - Depopulated Parts

Qty.	Refdes.	Description
2		Transistor, SOT-323
4	R1, R2, R3, R4	Resistor, 0603
+/-15V Regulator		
3	C99, C109, C110	Ceramic Capacitor, 1210
2	C102, C103	Ceramic Capacitor, 0603
3.3-15V Adjustable Regulator		
2	C40, C45	Ceramic Capacitor, 1210
Microcontroller Subsystem		
1	R9	Resistor, 0603
Level Shifter Subsystem		
6	C55, C56, C59, C81, C82, C85	Ceramic Capacitor, 0603
6	R77, R78, R91, R195, R196, R209	Resistor, 0603
4	R110, R111, R228, R229	Jumper, 0603